Scaleable Code Clone Detection

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Niko Schwarz
von Deutschland

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Scaleable Code Clone Detection

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Niko Schwarz

von Deutschland

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 7.02.2014

Der Dekan:
Prof. Dr. S. Decurtins

Acknowledgments

I'd like to dedicate this thesis to my wife, Katja Schwarz.

All results presented in this thesis are the result of collaboration. I have been helped in various ways by many people, including Ed Tavinor, Adrian Kuhn, Toon Verwaest, Erwann Wernli, Alexey Kolesnichenko, Aaron Karper, Simon Vogt, Nicole Haenni, Cedric Reichenbach, Chenglin Zhong, Muyao Zhu, Mircea Lungu, Romain Robbes, Yingnong Dang, Serge Demeyer, Irina Todoran and Matthias Zwicker.

All results shown in this thesis are the result of my work at the Software Composition Group, at the University of Bern, under the wise council of Oscar Nierstrasz. It was his publications that attracted me to switch from Mathematics to Software Engineering. Rarely did I choose so wisely. I am immensely grateful for the opportunities he gave me by letting me join his group.

I feel that I owe special thanks to Michael Conradt and Lars Clausen. They taught me many of the few things that I understand about computer science during my internship at Google Munich in 2012.

I'd like to thank Michael Godfrey for refereeing this thesis and accepting to be on the PhD committee.

Abstract

Code clone detection helps connect developers across projects, if we do it on a large scale. The cornerstones that allow clone detection to work on a large scale are: (1) bad hashing (2) lightweight parsing using regular expressions and (3) MapReduce pipelines.

Bad hashing means to determine whether or not two artifacts are similar by checking whether their hashes are identical. We show a bad hashing scheme that works well on source code.

Lightweight parsing using regular expressions is our technique of obtaining entire parse trees from regular expressions, robustly and efficiently. We detail the algorithm and implementation of one such regular expression engine.

MapReduce pipelines are a way of expressing a computation such that it can automatically and simply be parallelized. We detail the design and implementation of one such MapReduce pipeline that is efficient and debuggable.

We show a clone detector that combines these cornerstones to detect code clones across all projects, across all versions of each project.

Contents

Chapter 1

Introduction

Detecting code duplication in large code bases, or even across project boundaries, is problematic due to the massive amount of data involved. It is attractive, since studying the code cloned between projects opens a window into developer collaboration. The clones of a sample snippet of documentation likely finds typical uses of a library. A slightly modified code snippet in another project may indicate a fixed bug in that snippet. Clones are connections between different projects. Scaleable code clone detection provides a platform for helping developers collaborate between projects.

Software clones are not typically exact copies of each other, but rather start as exact copies and then evolve in different ways, for a number of reasons in accordance with the specific needs in their respective environments [49, 45].

Our approach follows the following scheme. An artifact is reduced to a hash, and then an entry in a big table is stored, which maps that signature to its origin. This storage scheme scales very well.

Since every artifact has at least one binary representation, the simplest signature is a hash of that representation. To allow for similar but different things to be identified, the signatures are derived from abstractions, *i.e.,* reductions of the original artifact by stripping away minutiae [24] that are not important for overall similarity. In this thesis, we show an abstraction that maps similar code snippets to the same hashes. We call the technique mapping similar artifacts to identical hashes *bad hashing*.

To eliminate the minutiae of source code, we must parse, which can be very expensive. However, while regular expressions cannot fully parse source code, they can approximate a correct parsing remarkably well, at a fraction of the cost of full parsing, while simultaneously producing simpler parse trees. The technique of producing parse trees from a single regular expression, we call *Lightweight parsing from regular expressions*.

MapReduce pipelines are a way of expressing a computation such that it can automatically and simply be parallelized. We detail the design and implementation of one such MapReduce pipeline that is efficient and debuggable.

This thesis suggests an algorithm to detect code clones across all Java projects, across all versions, using the above ingredients. The algorithm is *scaleable*, that is: it can handle growing amounts of work, simply by enlarging the system [9].

We can now state our thesis as follows.

Thesis:

Code clone detection helps connect developers across projects, if we do it on a large scale. The cornerstones that allow clone detection to work on a large scale are: (1) bad hashing (2) lightweight parsing using regular expressions and (3) MapReduce pipelines.

1.1 Motivation

Since the rise of internet-scale code search engines, searching for reusable source code has become a fundamental activity for developers [70]. Developers use search engines to find and reuse software. The promise of search-driven development is that developers will save time and resources by using search results. However, there are perils: the current practice of manually integrating code search results into a local code base leads to a proliferation of untracked code clones. As a side effect, bugs fixed in one clone typically do not traverse their new environment anymore, and the same holds true for extensions and code cleanups.

Even if they appear in the same project, code clones often cannot be eliminated [49]. But oversights in consistently applying changes to clones may introduce bugs into the system [29]. Therefore, tools have been proposed to maintain links between code clones [39, 29, 89], but they fail to link clones that are beyond project boundaries. Codebook by Begel *et al.* [5], is a social network in which people can befriend both other people and the artifacts they produce. Codebook is intended to maintain links between clones, it is however unclear how these links come into being. Begel *et al.* only vaguely suggest how that should be done: edges are to be added between a definition and its likely clones.

This thesis is motivated by a scheme to initially create and then maintain such links, called *hot clones*. While we have not implemented it in full, it shows that adding scaleability to code clone detection leads to opportunities that were previously unavailable. Thus, we view code clone detection as a *platform* to connect developers across different projects.

Hot clones are to be used as follows. A code search engine assists the developer by integrating its results into the source code. The IDE then remembers the origin of the code snippet and informs the repository that a clone was created, thus creating a link between original and copy. We will refer to clones created in this way as hot clones. Whenever a hot clone changes, the linked clones' developers are informed and offered the option to update their instance. Also, whenever a method is inspected, its clones can be inspected too, providing valuable information. The connections between clone instances are thus proactive and bidirectional.

Since software diverges, it is important to search across all versions of known software projects, in order to detect a hot clone. Once the same snippet was found in two different places, it can be traced in both places individually, in order to provide valuable feedback to the authors on both ends [77]. All methods that contain the snippet are added to the hot clone. All future versions of the method, whether or not they contain the original snippet, are now tracked in the hot clone. Since this is possibly too inclusive, it is vital to give the user the option to cut off the tracking of hot clones.

In the terminology of Koschke [51], this provides *compensative clone management*, *i.e.,* we limit the negative impact of existing clones, but we also give developers benefits from the software duplication introduced by clones by providing developers with information on how their code is used and modified.

The idea of hot clones scratches two itches. The first is to let developers benefit from code cloning, and the second is to provide researchers with more information on how cloning is used.

We believe that hot clones can ease backporting changes that occur in a linked clone. We believe that during development, hot clones will provide important feedback to developers. Contrasting one's own code with modified clones will give hints to bugs in related code, usage patterns, and plain examples of usage.

The nature of clones within a single software project has previously been studied [45], but the evolution of code snippets copied from searches in software projects has not. A prototypical implementation of hot clones would provide this opportunity. Being able to track the further evolution of code snippets after they are copied out of a search engine may give us great insight into the evolution of code, beyond the classification provided by Kapser and Godfrey [45]. If used by only a few developers, hot clones can provide insights from both a larger set of data than before, and from a wider range of uses.

In this dissertation, we show a scaleable clone detector that creates hot clones but is not yet integrated into development tools.

1.2 Applicability

Bad hashing and hot clones may apply to more than source code. For example, reducing the resolution of a bitmap image is a simple way of abstracting away minutiae that are unimportant for overall similarity (albeit this step alone is probably insufficient).

This scheme is applicable in a much wider context. We can store the signatures of all digital artifacts ever produced, in all of their versions. It maps from the signature to a descriptor of the origin of the artifact. This allows one to track the divergent evolution of software artifacts, as well as establish the provenance of every artifact encountered.

A big database of signatures mapping to sources represents an invaluable source for future research, going even beyond our use case of hot clones. As a rule of thumb, with massive amounts of data to help, difficult problems can suddenly become a lot easier [35]. The following are examples of where we think a big database of signatures would be valuable.

- *License compliance.* An organization needs to identify the origin of the software they create (either locally created or licensed) in order to verify that it has satisfied any legal obligations.

- *Security.* The origin of a copy is likely to continue to evolve. It is important to know if any of the copied artifacts contain security related bugs that have been fixed after the copy was made.

- *Verification of binaries.* A customer who subcontracts somebody to develop a software system might have received both source code and bina-

ries. In this scenario, the customer might want to verify that the binaries provided come exactly from the provided source code.

- *Plagiarism.* The owner of the original artifact might want to verify if a copy has been improperly made. In this scenario, the owner might want to find copies of her software artifacts.

1.3 Outline

This thesis is structured as follows.

- In chapter 2, we give an overview of existing clone detection techniques. The differences between our work and existing regular expression engines and MapReduce pipelines are presented inline in the relevant chapters.
- In chapter 3, we report on an empirical study to gather the actual information needs of developers across repositories. We conclude that code clone detection across repositories can help satisfy some of the information needs.
- In chapter 4, we detail our approach to bad hashing, and report on an empirical study on the prevalence of code clones across repositories. We conclude that code clones are common enough to serve as links between projects.
- In chapter 5, we show a technique to extract parse trees from regular expressions. We show that full parse-trees can be produced from matching regular expressions against input text. We conclude that this can be used to robustly approximate full parsing of source code in linear time.
- In chapter 6, we show a MapReduce pipeline that makes large-scale software easier to test and write.
- In chapter 7, we detail our approach to large scale clone detection. We show how it combines bad hashing, lightweight parsing using regular expressions, and MapReduce pipelines.

Acknowledgements

The conceptual design of hot clones was the result of a collaboration with Adrian Kuhn and Erwann Wernli [79].

Chapter 2

Related work

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 [73]. Bellon *et al.* categorize code clones as follows [8]. Type-1 clones are defined as "identical code fragments except for variations in whitespace, layout and comments". Type-2 clones are defined as "syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments" 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".

Cloning is common. 19% of the X Window System are clones[3]. Mayrand *et al.* found that between 6.4% and 7.5% of a large telecommunication system are type-1 clones. Jarzabek and Shubiao[43] go so far as to consider 68% of the Java Buffer Library clones.

Cloning is common if code needs to be modified that one doesn't own [23]. Rosson and Carroll [72] notice that sample code is used as a quick-start. Developing programs as modified clones of other programs is reported to be attractive in the financial industry[17]. Rysselberghe and Demeyer [91] show that even simple line-based clone detection is effective for refactoring. Rieger[71] gives an overview of the tradeoffs that cloning entails.

The quality of a clone detector can be measured in precision and recall [7]. Precision measures the fraction of true clones among reported clones. Recall measures the fraction of true clones that a detector can find.

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.

2.1 Index-based detectors

Index-based approaches, first suggested by Hummel *et al.* [40], save computation by not having a clustering stage. 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.

Keivanloo *et al.* [48] 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. Furthermore, 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.

2.2 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 [44]. Livieri *et al.* [59] 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.* [90] show how simhashes can speed up 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.* [53] 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 [15]. 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.* [24] 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.

2.3 Techniques

The following techniques are helpful for detecting clones.

Broder [11] 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 [26].

The idea of using bad hashes for clone detection was proposed by Baxter *et al.* [4]. Their approach creates bad hashes for sub-trees of the ASTs of classes, and thus requires full parsing of the source code in question.

2.4 Challenge

We distill as a challenge for a clone detector the ability to scale to arbitrary amounts of source code, while achieving good precision and recall.

Chapter 3

Information needs across projects

To build tools that use code clone detection to connect developers across different projects, we need to understand the information needs of developers with respect to other projects. A common relationship between the systems in a software ecosystem is that of *reuse based dependency*: a library or framework (the *upstream*) provides the required source code to another project (the *downstream*).

We present the results of an investigation into the nature of the information needs of software developers across projects. In an open-question survey we asked framework and library developers about their information needs with respect to both their *upstream* and *downstream* projects. We investigated what kind of information is required, why is it necessary, and how the developers obtain this information.

We show that the downstream needs fall into three categories roughly corresponding to the different stages in their relation with an upstream: selection, adoption, and co-evolution. Upstream needs fall into two categories: project statistics and code usage. We show that to satisfy many of these needs developers use non-specific tools and ad hoc methods.

The needs are identified by means of an open-question survey[56]. The underlying research questions are:

(RQ.a) What are the information needs of a software developer working in an ecosystem context?

(RQ.b) Why is this information important to know?

(RQ.c) How do developers obtain this information?

The remainder of the chapter is organized as follows: In section §3.1, we describe the methodology for this study which comprises both an open-question survey and a closed-question online questionnaire. In section §3.2 we report on the identified information needs, by listing and categorizing them, and by showing a representative quotation to explain each. In section §3.3 we corroborate and correct the identified information needs by presenting the support in

terms of agreement on Likert items. In section §3.4, we discuss our methodology, research results, and highlight some possible future research directions. In section §3.5, we show how our work fits in the vein of previous empirical studies of developer needs. In section §3.6, we draw conclusions.

3.1 Research Method

To identify the needs of developers across projects, we interviewed several via email and in person. We asked the respondents what their information needs were corresponding to their upstream and downstream roles in the software ecosystem in which they craft software.

In table 3.1, we list the survey questions. Our research question (RQ.a) is split into questions 2 and 3 addressing respectively framework and library developers, and developers depending on code from other projects. Questions 2 and 3 are further divided into three subquestions each, asking *what* kind of information they need (RQ.a), *why* this information is important (RQ.b) and *how* they obtain that information at the moment (RQ.c).

To analyze the answers we received as free-form text, we applied a grounded theory methodology as introduced by Strauss and Corbin [84]. In this methodology, contrary to beginning with a pre-conceived theory, one is evolved from the data, and continuously refined in an iterative process. The data analysis includes coding strategies by breaking down the data collection from surveys or other observations into similar units. The questions are formulated as openly as possible and do not attempt to influence the participant in a certain direction.

We label each mentioned topic with an assigned code[1] This process of qualitative data analysis is known as open coding [12]. By grouping similar codes together we create axial coding categories and themes [84]. For each grouping, we provide a one-sentence description. The results of the open coding and of the axial coding are presented in section §3.2.

To triangulate the qualitative results we run a follow-up study in which we verify our results in a closed-question survey, where the participants do not answer with free text, but instead agree or disagree with the hypotheses we present in section §3.3.

3.2 Qualitative Results

In this section, we present a list of codes that resulted from the open-coding process. They represent the information needs, motivations, and current practices of software developers working in an ecosystem context.

We shipped the email survey to a convenience sampled [60] group of 20 framework and library developers. The participants were neither offered nor given compensation for their participation. Participants were assured of their anonymity.

Of the developers asked, 65.0% responded. An additional participant gave us the answers in person. All participants have at least seven years of academic or professional experience. From the 14 answers, we collected initial codes to answer

[1] The term "code" refers to a recurring topic in the interviews, and should not be confused with "source code."

1. In what ecosystem are you most active?
2. Are you the developer of a framework or library?
   If so, what is its name?
   - 2.a What do you most want to know about the use of your library/framework in your ecosystem?
   - 2.b Why would that be interesting to know?
   - 2.c What do you currently do to obtain that information, if anything?
3. Are you using a framework or a library in your ecosystem? If so, name one.
   - 3.a What do you most want to know about the libraries/frameworks that you are using?
   - 3.b Why would that be interesting to know?
   - 3.c What do you currently do to obtain that information, if anything?

Figure 3.1: The survey as shipped to the participants.

the questions in table 3.1. We assigned each participant a reference letter from A through N. We explain our findings with quotations of the participants with the corresponding reference letter.

The goal of the first question was to put the respondent into the right frame of mind in which he would think about the broader context of his work and the inter-dependencies of the systems he is working on. We do not analyze these answers here, but let us mention that we had a variety of ecosystems centered around different languages (Smalltalk, Python), technologies (Moose, SciPy), and online source code repositories (SqueakSource[2], Github[3]). Two respondents mentioned two social websites (StackOverflow[4] and Reddit[5]).

The notation we use is as follows. Every discovered code gets an identifier (*e.g., UN-1*), denoting the anonymized participants that named it. We then explain the definition of the code, and finally give a representative quote of the code [56, p. 284].

The survey aims at capturing information needs that the developers of libraries and frameworks have. To corroborate the identified need, we ask for

[2] `http://www.squeaksource.com`
[3] `http://www.github.com`
[4] `http://stackoverflow.com`
[5] `http://reddit.com`

their motivations and current practices. We will list and discuss the codes we extracted for each of the questions individually.

3.2.1 Upstream needs

This section discusses the information needs of upstream developers.

Code Usage

This grouping holds developer needs that detail how people use source code.

UN-2. **API usage details.** (B,F,J,K,L) Developers monitor the way the downstream is using the API and collect details about invoked methods and their arguments. This provides insight into the effectiveness of an API and its usage: *"L: which parts of the code are actively used?" (L).* A respondent confirms: *"I like to know how people are using my code in order to make the framework better. It's not just about minimizing the impact of changes, but also about seeing what's awkward, what features are used in conjunction and which independently, which areas are performance sensitive* etc.*" (LS-57, UN-2).*

UN-4. **Runtime statistics.** (B) Some developers want statistics about the usage of their library at runtime to help localize and fix failures: *"which API methods are called how often and which data is passed to them? How often do they fail with an error?" (B).*

UN-5. **Code convention compliance.** (E) This includes naming conventions, indentation, comments and so on. A guideline would provide help for maintenance issues, consistency and readability. *"Variation of lint rules in my projects along the project history" (E)* to ensure that downstream developers follow the conventions the developer set.

Project Statistics

This grouping holds the need for simple, descriptive numbers describing the project.

UN-1. **Downstream projects.** (A,C,D,F,I) Developers want to know how their code fits in the ecosystem. They want to know the number and nature of their downstream projects, and for what purposes the downstream is using a project: *"I'd like to know what people build with my frameworks" (A).* Respondent (D) wanted to know number of passive downstream developers that track a project's state. A respondent states *"if people are making downstream fixes it would be helpful to know this. So [the code changes] can be merged." (LS-42, UN-1/UM-3).* :w

UN-3. **Forked projects.** (D,J,L) Developers want to know about the clones of their work. With infrastructures like Github this is particularly easy to do.

3.2.2 Upstream Motivations

This grouping holds the answers to question 2.b, representing the motivation behind the stated needs.

UM-1. **Strengthening self-esteem.** (A,I,J,K) Pride in one's work and project motivates information needs. *"It is a good motivation if a lot of people like my code and build cool stuff on top of it" (A)* and *"it helps the self-esteem" (A).* Positive feedback and rising popularity keeps a developer motivated and *"gives inspiration and hints where to orient the project's evolution" (K).*

UM-2. **Maintaining downstream compatibility.** (F,I,K) When developers know how their clients use their framework or library, then they are able to estimate the impact of code changes. If needed, they can notify downstream developers on how to stay compatible. A participant explains: *"I want to know [. . .] the impact [. . .] when I modify my source code" (K).* And another developer states: *"I want my clients to know how the library is being used and to assess the impact of possible changes" (F).*

UM-3. **Managing resources.** (B,L) Discover unused functionalities to deprecate them out and to better distribute effort. *"To conserve my resources. If people don't use a method or, a whole feature of the API, why maintain it?" (B)*

3.2.3 Upstream Practices

This grouping holds answers to question 2.c, representing current practices upstream developers use to fulfill their information needs.

UP-1. **Mailing lists.** (A,F,I,J) People with common interests subscribe to a mailing list to keep up-to-date with a given issue. Problems and solutions are asked and discussed through email communication with all subscriptions.

UP-2. **Repository analytics.** (A,C,D) Some source code repositories provide analytics for projects. GitHub provides information about forks, downloads, watches, *etc.* Even monitoring web traffic is of interest: *"I observe the web analytics of my project's home page" (A).*

UP-3. **Monitoring ecosystem commits.** (F) In some cases developers track code changes to many projects of interest at once by monitoring news services: *"I am monitoring the RSS of SqueakSource [NB: which includes updates on the changes to several hundreds of active projects]" (F).*

UP-4. **Social media.** (A) Developers use social media tools (*e.g.,* Twitter) to publish the latest news about their project.

3.2.4 Downstream Needs

This section discusses the information needs of downstream developers.

Selection

This grouping holds the information needs of a developer during the process of selecting an upstream.

DN-2. **Available public support.** (A,B,E,J) Developers want to know the popularity of a framework *"Are they popular enough to find support on the web in blogs and on StackOverflow?" (B).*

A related factor is the responsiveness of the developer team and associated community to provide support: *"How likely are they to fix bugs and to respond to feature requests" (B). "... whether there are bugs that were left unresolved for a long time" (E). "As a developer (and user in certain cases), I want to be certain that the community is friendly, accepts [newbies] and responds fast." (LS-48)*

DN-4. **License type.** (A,I,L) A common request is: *"Is the license compatible with ours?" (A).*

DN-5. **Implementation quality.** (B,E) A potential client of a library wants to know how robust its implementation is, how often it is updated, how responsive the developers are, and how fast the library is evolving. *"Whether [the project's code] works or not" (E).*

People want to know the level of activity around a library: *"Whether they [the libraries] are intensively maintained" (B).*

DN-8. **Comparison with similar upstreams.** (A) Find related libraries and frameworks that provide similar functionalities but are independently developed. *"Comparison with similar frameworks" (A)* gives the opportunity to consider an alternative upstream.

Adoption

This grouping groups the information needs that pertain to a developer starting to work with a new upstream.

DN-3. **Documentation.** (B,G,H,N) The potential users of an API require its documentation: *"I am basically happy with a good API documentation" (B).*

Some developers want to understand the internals of an upstream project and thus require architectural documentation: *"[...] expose connections between high-level elements [...] what methods [...] of the packages invoke each other" (N).*

DN-7. **Real contextual sample code.** (C) Developers want example code snippets that are extracted from other projects with similar functionality. *"I'd like to see example code extracted from other projects using the same libs that corresponds to functions I'm trying to figure out how to use" (C).*

Co-evolution

This grouping holds the information needs that arise from developers' project co-evolving with others.

DN-1. **Upstream changes.** (E,F,K,N) Developers want notifications of deprecations and substitutions that affect the API they use: *"What has changed since the last time I loaded [the library]" (K)* and *"if they deprecated some methods" (N)*

They might also care about the developers that make the changes: *"who changed what" (F).*

Finally, when developers have a portfolio of projects, they care about how a third-party upstream impacts it: *"Which of my projects may be impacted by some update of Pharo" (E).*

DN-6. **Compatibility with other systems.** (L) A downstream client often depends on multiple upstreams. They want to know whether an individual upstream works with the rest of the configuration. *"Does the current version [of the upstream] run on the version of the system I use [downstream]" (L).*

3.2.5 Downstream Motivations

This grouping holds the answers to question 3.b, representing the motivation behind the stated needs.

DM-1. **API understanding.** (C,F,G,I,L,M) Developers want to use functionalities provided by the API right away. This is eased when API names are intuitive and well documented. *"To see whether I can construct on the libraries or not" (F).* A participant's answer is that he would like *"to spend less time figuring out how to use new libraries" (C).*

DM-2. **Keeping up with upstream evolution.** (E,I,J,L)

Developers of downstream projects want to keep up to date with upstream changes. The only way to improve something is to know the existing problems and to know how it is expected to work (I). *"To know whether I have to update my projects or not" (E)*, *e.g.,* if there are any new releases. The same respondent correlates to the credibility of the upstream: *"I am interested to see if the change was performed by someone I trust" (E).*

DM-3. **Choosing the right upstream.** (B,I) Choosing the right upstream will impact the future of a project: *"For example, [our testing framework] uses JUnit 4, but later I learned that less than 5% of all users of JUnit use version 4 and all others still use version 3. So we are stuck with a bad choice" (B).* Another developer argues: *"... if I don't know how to use [the library] after an hour, I throw it away. I won't look one single day into its code just to see how to use it" (I).*

DM-4. **Influencing upstream.** (B,J) Sometimes developers would like to modify the upstream to conform to their needs, but this is not always possible: *"Sometimes I need to collaborate and influence design of frameworks I use and to ensure I can progress even if the maintainers I depend on are not responsive" (J).*

DM-5. **Estimating the impact of changes.** (F,H) Before updating to a new version of the upstream, developers want to estimate the impact of changes. They are *"interested in what the change affected" (F).*

3.2.6 Downstream Practices

This grouping holds answers to question 3.c, representing current practices downstream developers use to fulfill their information needs.

DP-1. **Monitoring news.** (C,E,F,G,I,J) Developers read mailing lists and monitor repositories for commits and activities to be up to date. Developers monitor the RSS feeds of the upstream projects: *"I am monitoring the RSS of SqueakSource" (F).*

DP-2. **Searching the Internet.** (A,B,C,G,H,I) Downstream developers search the internet for the upstream developer's website or third parties blogs and tutorials. Before using a specific framework, downstream developers like to play around and modify example code to see how it works.

Developers often estimate the relevance of a library by its popularity online, and in programming related forums. *"I look at the most popular tags on Stackoverflow and pick that library" (B).*

DP-3. **Continuous integration.** (F,K,L) Some developers commit code changes to the project repository several times a day. As one respondent states, *"I am building regularly to ensure that at least things still work" (F).* This supports fast deployment and uncovers compatibility problems in early stages.

*DP-4.***Unit tests.** (E) I load the latest upstream version and run my unit tests.

3.3 Quantitative Results

We continued our study by verifying our results in a closed-questioned online survey[6]. This time, the participants do not answer with free text, but instead their answers range from full disagreement to full agreement on a series of numerical 5-point Likert items [58, 56]. The qualitative results from section §3.2 serve as an initial position to formulate suitable statements. For each code, we formulate at least one statement. If applicable, we used quotations from the participants.

A total of 68 Likert item questions were asked; 32 questions (Q1.1–Q3.5) to upstream developers and 36 questions (Q4.1-Q6.7) to downstream developers. Furthermore, we asked several pre-survey questions about developer background and three voluntary open-ended post-survey ones.

After testing a pilot version of the survey, we advertised the survey in various mailing lists: Open JDK, Processing.js, jQuery, CakePHP, SciPy, NumPy, Pharo, Squeak, Seaside, Drupal, Coreaudio, Apache Hadoop, Apache Cassandra, Google WebToolkit, Ubuntu, Soot and Zend Framework.

We received 75 responses, 46 were framework and library developers and 29 were framework and library users. Even though no response rate could be determined, we reached participants across the world (46% from Europe, 32% North America, 8% from Asian, 6% from South America, 4% from Australia and 1% from Africa).

[6]The exact survey is available at `http://goo.gl/q2ABRd`

The professional experience of respondents is distributed as follows: 29% have 5-10 years, 22% have 11-20 years and 22% have more than 20 years. 17% have less than 5 years experiences and 10% have not answered.

3.3.1 Upstream Information Needs

Let us discuss the needs identified in the qualitative analysis in section §3.2.

Code Usage

The responses to our questions on code usage-related information needs can be seen in figure 3.2.

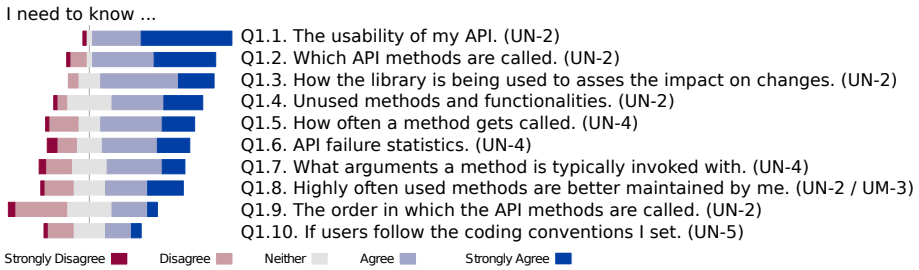


Figure 3.2: Downstream code usage.

The information need with the highest confirmation rate was "API usage details" (UN-2). The most agreed upon statement for developers was that they want to know the usability of their API. The next was more detailed: what methods are called by users followed by what methods are not used.

The importance of "Runtime Statistics" (UN-4) is less supported by developer feedback. The developers that care about this, care about API failures and the type of data that flows through their programs.

The least agreed upon need is code convention compliance (UN-5).

Project statistics

The responses to our questions on project statistics-related information needs can be seen in figure 3.3.

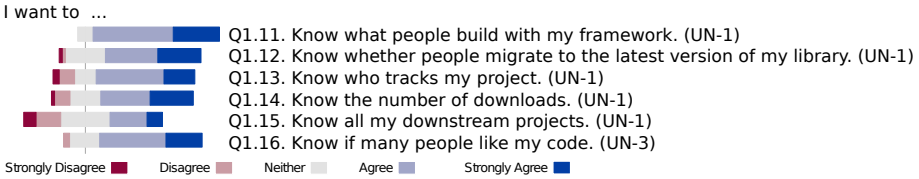


Figure 3.3: Downstream project statistics.

The information need "downstream projects" (UN-1) received strongly positive feedback. The most-agreed-upon statement was that developers need to know what clients build with their code.

3.3.2 Upstream Motivations

The responses to our questions on code upstream motivation can be seen in figure 3.4.

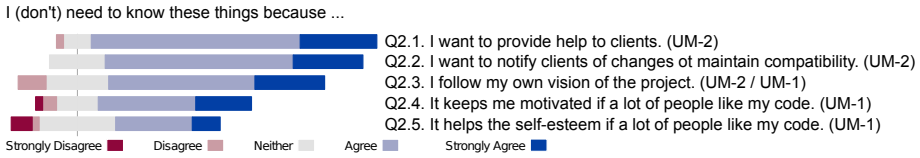


Figure 3.4: Upstream motivations.

The motivation "strengthening self-esteem" is slightly less supported than "maintaining downstream compatibility", though both are strongly supported.

3.3.3 Upstream Practices

The responses to our questions on practices can be seen in figure 3.5.

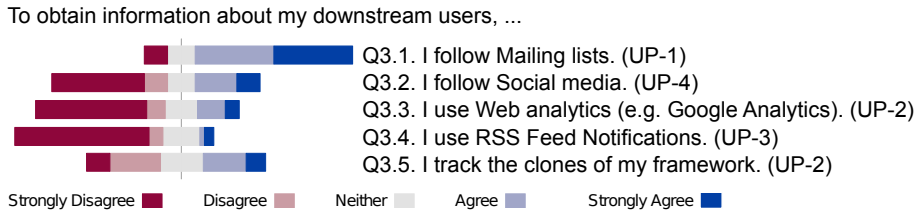


Figure 3.5: Current practices by upstream.

The statement that got the strongest support was that developers follow mailing lists in order to learn about the way their code is used. The other statements received generally negative feedback, indicating either that we overlooked existing tools and practicing, or that such tools do not exist.

3.3.4 Downstream Information Needs

In this section, we address all developers that work in a *downstream* context. *Upstream* developers were also asked to take part in the role as a *downstream* developer.

Selection

The response to our questions on selection-related information needs can be seen in figure 3.6.

For the information need "Implementation quality" (DN-5), developers agree that knowing whether or not a project works is important for selecting it.

The information need "Available public support" (DN-2) is mostly supported, but we see that respondents are discordant as to whether or not it is important to know who the developers are. The information need "Comparison with similar upstreams" (DN-8) is strongly supported. The information need "License type" (DN-4) is strongly supported.

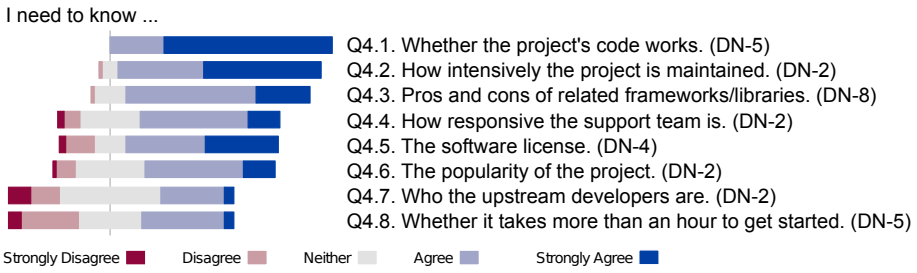


Figure 3.6: Downstream selection.

Adoption

The response to our questions on selection-related information needs can be seen in figure 3.7.

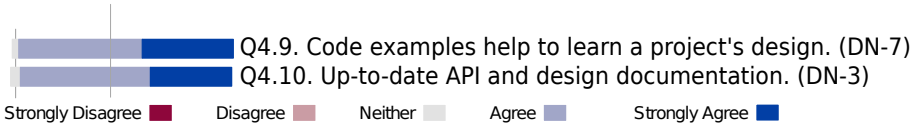


Figure 3.7: Downstream adaption.

The information needs "Documentation" (DN-3) and "Real contextual sample code" (DN-7) are strongly supported. None of the respondents disagreed with either of the two statements regarding code examples and API documentation.

Co-evolution with an upstream

The response to our questions on co-evolution-related information needs can be seen in figure 3.6.

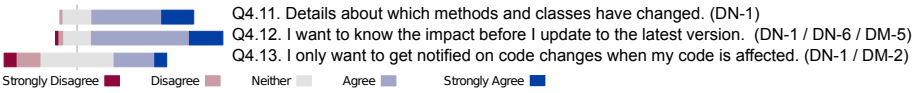


Figure 3.8: Downstream co-evolution.

The information need "Upstream changes" (DN-1) is supported. Developers are interested in details about which methods and classes have changed and whether these changes have an impact on their own source code.

3.3.5 Downstream Motivations

The response to our questions on motivation can be seen in figure 3.9.

Our identified motivations "Choosing the right upstream" (DM-3), "API understanding" (DM-1), "Keeping up with upstream evolution" (DM-2) are all inconclusive.

Motivation "Estimating the impact of changes" is supported. Developers agree that they avoid code adaptation if the estimated time is excessive.

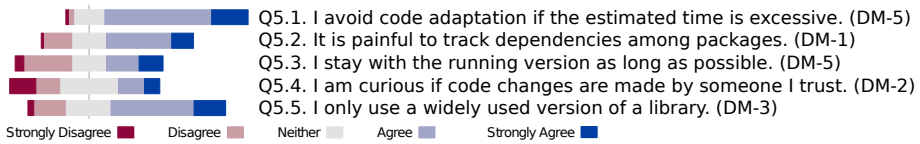


Figure 3.9: Downstream motivation.

3.3.6 Downstream Practices

The response to our questions on selection-related information needs can be seen in figure 3.10.

Figure 3.10: Current practices by downstream.

Current practice "Monitoring news" (DP-1) is supported. Developers subscribe to mailing lists. Current practice "Searching the Internet" (DP-2) is strongly supported. Developers routinely search for blog posts and tutorials. Current practice "Continuous integration" (DP-3) is supported. Developers routinely run integration tests. Current practice "Unit tests" (DP-4) is inconclusive.

3.4 Discussion

Threats to validity. Since our questionnaire does not use balanced keyin [16], and therefore is subject to acquiescence bias. This hinders any further statistical analysis, since we cannot separate acquiescence bias from actual agreement. We therefore attempted no aggregation over Likert items. We have not confirmed the that the Likert items we chose operationalize the information needs that we are interested in. Since our test population was convenience sampled, the generalizability of our study is limited.

Our qualitative research method based on grounded theory does not guarantee completeness of our results [56]. Most of the results depend on the selected participants and their opinion and experience.

Tools currently addressing information needs. We identified two categories of information needs of upstream developers: project statistics and code usage. Today's tool support and practices are well-supported with respect to the first need (project statistics) but fail to support the second need (code usage).

Version control system and hosting platforms like GitHub offer functionalities to get information about number of downloads, number of followers [22]. However, information needs pertaining to API usage are ill-supported. Tools should offer method and parameters calls and function changes.

Code clone detection. Code clone detection on a large scale can directly answer many of the identified information needs. Finding exact duplicates of an entire library indicates that the library is used as-is in another project, yielding an important *run-time statistic*. The number of clones inside of a project may be indicative of the *implementation quality*. The most frequent clones of all are licenses. This gives us a complete list of all commonly used license templates. Finding their clones in all projects allows us to auto-detect the *license* of all projects.

Finding type-3 clones with the documentation of a project is likely to find projects that used the code samples from the documentation as a starting point. This gives *sample code*. *Real, contextual sample code* is found in the type-3 clones of the call sequences of the methods of a library. Knowing from clone detection where a library was taken from allows us to inform developers of *upstream changes*.

Even more importantly, scaleable code clone detection can be used as the platform for more specialized tools addressing these needs. For example, finding call sequences may need to use another way of bad hashing than finding edited copies of text.

3.5 Related Work

Sillito *et al.* [82] identify 44 questions developers ask when changing a software task. They are specific to their single-system task and refer to implementation details (*e.g.,* method calls, data structures, type hierarchies). Four categories could be found. They assume the project's source code can be represented as a graph; with software artifacts as nodes and references or relationships as edges.

Ko *et al.* [50] conducted a study in finding information needs in development teams. They collected their data by recording questions developers posed during their daily programming tasks. Their findings include 21 different types of information in seven categories. The majority refer to knowledge of software artifacts or co-workers.

Seichter *et al.* [81] examine an information retrieval management system for software artifacts to improve collaborations. Inspired by social networks, they propose different kinds of relationships to interconnect software artifacts within a software ecosystem. They define types of interactions but do not declare specific information needs.

Begel *et al.* [6] propose Codebook as a code-based social networking web service that helps developers get information about other activities of their colleagues. They asked programmers inside Microsoft company about inter-team collaboration problems. They identified and grouped 31 information needs into eight categories: "change notification, finding dependents, finding other people, finding artifacts, awareness of other teams, artifact history, working planning and social networking".

They discovered that programmers are often interested in finding the people responsible for certain parts of the code base. In our case, the respondents were less interested in who wrote the code, but more interested in its quality and functionality. Their solution, Codebook, serves as a social networking tool to connect developers together with software artifacts.

Phillips *et al.* [66] identify information needs to integrate branched version of a software project. They found four needs: Identifying conflicts before they arise, monitoring features with their dependencies, tracking measured data about number of bugs, test results *etc.*.

In contrast, our study studies the information needs of upstream and downstream developers across projects.

3.6 Conclusions

Our study suggests that upstream and downstream developer needs in a software ecosystem are different and that the downstream needs are more numerous.

The following information needs, motivations behind them, and current practices to alleviate them were identified, and found support in quantitative analysis: Upstream developers wish to see "Runtime Statistics" on their code. They are interested in their "downstream projects". Upstream developers' information needs are motivated by "strengthening self-esteem", and "maintaining downstream compatibility" Currently, they satisfy their needs by "following mailing lists".

Downstream developers were found to have a different set of information needs, motivations, and practices. About the code and projects they depend on, they wish to know "Implementation quality", "Available public support", "Comparison with similar upstreams", "License type" Downstream developers, when adopting a new dependency, need to know "Documentation" and "Real contextual sample code". In the face of co-evolving dependencies, they wish to be informed of "Upstream changes". Downstream developers' information needs are motivated by "Estimating the impact of changes". Current practices to fulfill their needs are "Monitoring news", "Searching the Internet", "Continuous integration", and "Unit tests".

Code clone detection on a large scale can directly answer many of the identified information needs. Where it cannot directly answer them, it can serve as a platform to build tools to answer them.

Acknowledgments

This chapter is based on work I did with Nicole Haenni, Mircea Lungu, and Oscar Nierstrasz [34].

Chapter 4

Bad hashing

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 *bad hashing* as a technique that may scale up to very large amounts of source code in the presence of multiple versions.

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.

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 [48]: 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.

Current hash-based approaches to clone detection handle only type-1 and type-2. In this chapter, 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 [49].

In this chapter, 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 chapter are threefold:

1. Three bad hashes that can be used to find cloned methods across an entire ecosystems.
2. An evaluation of the three detection techniques in terms of performance on a real-world software ecosystem which demonstrates their scaleability: running the analysis on 47 GB of source code takes hours.
3. We show that a large amount of code is duplicated, and that clone groups can feature hundreds of members across many projects.

4.1 Approach

To handle large amounts of data, we took Broder's [11] 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. We define the following bad hashes, one for each clone type.

4.1.1 Type 1: Hashes of source code

Type-1 clones are defined as "identical code fragments except for variations in whitespace, layout and comments" [8]. However, identifying comments is language-specific, and want to stay language-independent if possible. On the other hand, if we find a literally copied comment of substantial size, then this provides strong evidence of cloning having happened. It seems negligent to ignore the information. In our approach, we therefore do not ignore comments.

We define our bad hash such that two documents are detected as 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[1] of the resulting string. Then, two snippets are detected as type-1 clones *iff* they produce the same hash.

4.1.2 Type 2: Hashes of source code with renames

Type-2 clones are detected as "syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments" [8]. We use the following bad hash: Two documents 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 4.1.

[1] SHA1 hashes are cryptographic hashes. Cryptographic means that we can assume them, for all practical purposes, to never collide.

Table 4.1: 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 [10] and it is computationally inexpensive. While this definition appears to be inclusive, as we will see in Table 4.3, it catches barely more clones than there are type-1 clones.

4.1.3 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" [8]. This definition leaves open just how much "further modification" is tolerable; clearly, it appeals to the intuitive sense of similarity. Broder [11] 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 shingles selected to represent the document. The last definition hinges on selection of shingles into a *sketch* of a snippet. 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'.

We use the following bad hash. Two documents are detected as type-3 clones if and only if they have 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. Table 4.2 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.

While our definition of a type-3 clone is equivalent to Broder's Option B predictor with parameters $w = 4$, $m = 4$ [11, Theorem 1], it works differently.

Table 4.2: 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.

Normalized	Shingles	hashes
`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 t: t, t t: t t, t: t t. ^, t t. ^ t`	bd2d, c80b, a3f8, 11b5, 6951, 4f55, a43b, 8f58, f7d2, d549, bcee, fbe7, 84f4

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.

Our definition chooses an expected quarter of shingles by looking for those whose hash ends in the binary pattern '11'. This was done to give every source code modification a chance of being detected. If, for example, we were to demand that we keep at least one shingle per line, then all changes that lead to different shingles in all lines, for example by prefixing every line with `//`, are undetectable.

There is an obvious downside to our definition: since we define only one hash per method, we are incapable to detect clones at the sub-method level. Obviously, this limits the recall of this approach. We'll see a refinement in chapter 7.

4.2 Empirical Study: SqueakSource

We used our approach to detect code duplication across repositories on SqueakSource (`http://www.squeaksource.com`); SqueakSource was the de-facto standard code repository in the Smalltalk ecosystem. In June 2011, SqueakSource contained 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).

4.2.1 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.[2]

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 [59].

What makes our approach lightweight, in comparison, is the lack of a dedicated clustering stage. Defining similarity as an equivalence relation makes clustering trivial.

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 scaleability.

4.2.2 Clones in the SqueakSource ecosystem

Table 4.3 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

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

Table 4.3: 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 %

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 high chance of the sketch changing explains why our working definition of type-3 clones in section §4.1.3 clones is much more restrictive than it appears.

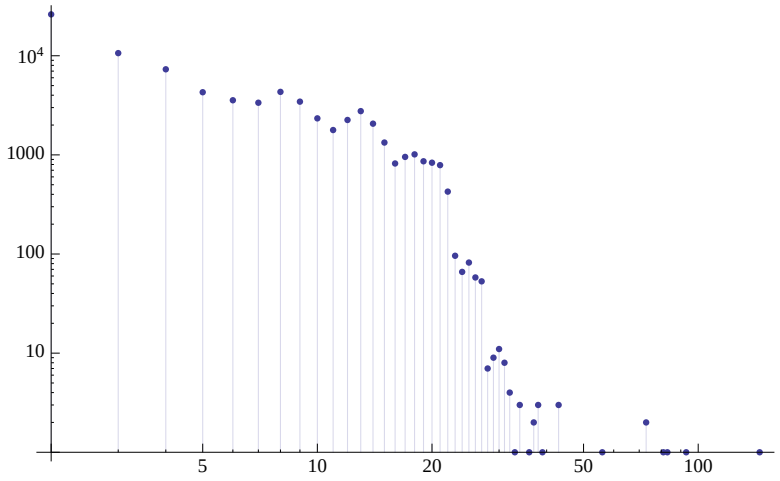


Figure 4.1: Distribution of clone group sizes for type 1 clones. The x-axis is the size of the clone groups; the y-axis is the number of clone groups of that size across the ecosystem.

Figure 4.1 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.

4.2.3 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 grows 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 [49], this approach underestimates cloning by more than 20%.

Since we look at all versions of all projects, we were interested to find out whether or not looking only at the latest versions of all projects would have sufficed. For every clone that we detected, in any version, we checked whether there is a corresponding clone in the latest version of that method. This serves as an indicator of how much cloning is missed by examining only latest versions. We found that 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.

4.3 Discussion

We have applied our techniques to a single ecosystem, which is comprised of Smalltalk projects only. Our findings may not generalize to other ecosystems, and other programming languages.

Our clone detection techniques function at the granularity of methods only. If two methods differ by more than two shingles, we are unlikely to detect them, even if they share long sections. This may have an impact on the type of clones detected; in particular, type 1 clones that are smaller than method boundaries may be classified as type 3 clones, or not detected at all. Therefore, our approach has poor recall for long methods. In chapter 7, we will see a way to use bad hashing that finds clones within methods.

Our clone detection technique has an important ingredient for scale: it avoids clustering. But we haven't yet described how it *scales*, that is, we have not yet described a scheme that handles increasing loads by simple growing of the system. Again, we will rectify this in chapter 7.

4.4 Conclusions

Even though classes are meant to be modular, we have found that methods are reused in new contexts far more frequently than classes.

Bad hashing simplifies the problem of finding similarities across projects, such that it can be solved on a large scale. 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 conclude that recall of clone detectors can be improved by examining more than the latest version.

Bad hashes, applied solely at the method level, may lead to poor recall. Despite that, 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. This implies that clones are common enough to serve as links between projects.

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.

Acknowledgements

This chaper is based on work I did with Mircea Lungu and Romain Robbes [78].

Chapter 5

Lightweight parsing using regular expressions

Regular expressions naturally and intuitively define parse trees that describe the text that they're parsing. While regular expressions aren't powerful enough to parse many grammars precisely, we can often find a regular expression that approximates a grammar reasonably well. For example, to split a Java source code file into classes, the following regular expression will do a reasonably good job: `\bclass\b.*?\{`, because `class` is a reserved keyword in Java. Somewhat unorthodoxly, the regular expression we gave is intended to match against the entire file, and match each individual class in a *capture group*, defined by the parentheses in the expression. Doing so will guarantee that the time to parse is linear in the size of the text input.

Lightweight parsing using regular expressions is linear in the size of the text input, and therefore indispensable to the scale of our clone detector. Lightweight parsing has another advantage: it is robust in the face of syntax errors. If a Java source code file cannot be compiled because of a syntax error, using a full Java parser would not just be expensive, it would also fail. As we will see when we describe the regular expression used by our clone detector in chapter 7, we will handcraft our regular expression to only identify the parts of the grammar of interest to us. If there is a syntactic error in the source code unrelated to our interests, it will be parsed correctly for our purposes, even if it is ungrammatical Java.

In this chapter, we describe a technique for building up the complete parse tree resulting from matching a text against a regular expression, where the nodes of the parse tree correspond to the repeated matchings of capture groups from regular expressions.

In standard DFA matching, all paths through the NFA are walked simultaneously, as if in different threads, where inside each thread, it is fully known when which capture group was entered or left. We extend this model to keep track of not just the last opening and closing of capture groups, but all of them. We do this by storing in every thread a history of the all groups using the flyweight pattern. Thus, we log enough information during parsing to build up the complete parse tree after matching.

A regular expression can easily describe that a text matches a comma sep-

arated values file, but it is unable to extract all the values. Instead it will only give a single instance of values: `((.*?),(\d+);)+` might describe a dataset of ASCII names with their numeric label. Matching the regular expression on `"Tom Lehrer,1;Alan Turing,2;"` will confirm that the list is well formed, but the match will only contain `"Tom Lehrer"` for the second capture group and `"1"` for the third. That is, the parse tree found by the POSIX is seen in Figure 5.1.

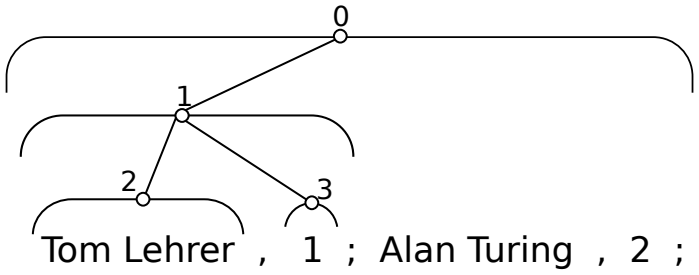


Figure 5.1: Parse tree produced by POSIX-compatible matching `((.*?),(\d+);)+` against input "Tom Lehrer,1;Alan Turing,2;".

With our algorithm we are able to reconstruct the full parse tree after the matching phase is done, as seen in Figure 5.2.

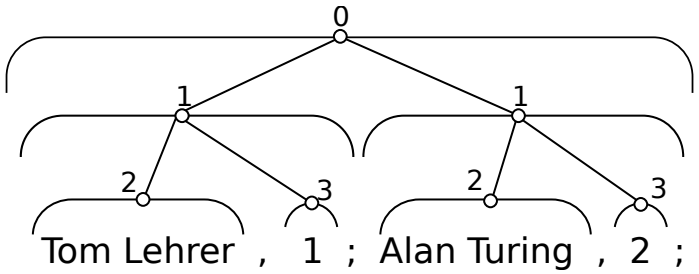


Figure 5.2: Parse tree produced by our approach matching regular expression `((.*?),(\d+);)+` against input "Tom Lehrer,1;Alan Turing,2;"

The amortized run time of our approach is $O(nm\log(m))$, where m is the length of the regular expression and n is the length of the parsed string. It is the first algorithm to achieve this bound, while extracting parse trees. The best-known algorithm to parse regular expressions, without extracting parse trees, run in time $O(mn)$[80].

5.1 More powerful than standard regular expressions

It may at first seem as if all capture groups can always, equivalently, be extracted by splitting the input, and then applying sub-regular expressions on the splits. This is, for example, an entirely valid strategy to extract the parse tree in Figure 5.2. However, this can quickly become an exercise of writing an entire parser, using no regular expression engine at all, even if the underlying grammar

is entirely regular. The following grammar is hard to parse using a regular expression engine, even though it is regular.

Consider a file of semicolon-terminated records, each record consisting of a comma-separated pair of entries, and each entry can be escaped to contain semicolons, as in the regular expression `(("[^"]*"|[a-z]*),("[^"]*"|[a-z]*);)+`. Here, expression `[^"]*` is a non-greedy match which will be discussed in more detail in Section §5.2. This language contains, for example, the string `"h;i",there;"h;,i",Paul;`. It is easy to see that, in order to extract all four capture group matches, it is insufficient to split the input at the semicolon, as that would split the field "h;i" in half. More involved examples, where hand-written parsers become harder to make, are easily constructed. In contrast, our approach yields the entire parse tree, simply from the regular expression.

5.2 Algorithm

Let us recall what DFAs (Deterministic finite automaton) and NFAs (Nondeterministic finite automaton). A DFA is a state machine that will walk over the transition graph, one step for every input character. The choice of transition is limited by the transition's character range. A transition can only be followed if the current input character is inside transition's character range. NFAs differ from DFAs in that for some input character and some state, there may be more than one applicable transition. If there is, an NFA will magically guess the correct one. Figure 5.3 shows an example of an NFA's transition graph. For the moment, let us discuss regular expression matching on NFAs. Assuming that the NFA just magically knows the right transition lets us focus on the important things, greediness control and capture groups.

Conceptually, our approach is the following pipeline of four stages.

1. Parse the regular expression string into an AST.
2. Transform the AST to an NFA.
3. Transform the NFA to a DFA.
4. Compactify the DFA.

In reality, things are a little more involved, since the transformation to DFA is lazy, and the compactification only happens after no lazy compilation has occurred in a while. Worse, compactification can be undone if needed. We'll get back to these details in section §5.3.3. Let's discuss the stages in turn, starting with 2, since step 1, the parsing of the regular expression grammar, is straightforwardly obtained by following the POSIX specification [69].

5.2.1 Thompson's construction

We transform the AST of the regular expression into an NFA, in a modified version of Thompson's NFA construction. To control greediness, or discern capture groups, our approach adds ε transitions to the transition graph. An ε transition has no input range assigned, and can thus always be used. It does not consume an input character. The additions are needed for greediness control and capture groups. Let's look at both, in turn.

To see the importance of greediness control, consider again the regular expression `((.*?),(\d+);)+`. The question mark sets the `.*` part of the regular expression to *non-greedy*, which means that it will match as little as possible while still producing a valid match, if any. Without provisioning `.*` to be non-greedy, a matching against input `"Tom Lehrer,1;Alan Turing,2;"` would match as much as possible into the first capture group, including the record separator ','. Thus, the first capture group would suddenly contain only one entry, and it would contain more than just names, namely "Tom Lehrer,1;Alan Turing". This is, of course, not what we expect. Non-greediness, here, ensures that we get "Tom Lehrer", then "Alan Turing" as the matches of the first capture group.

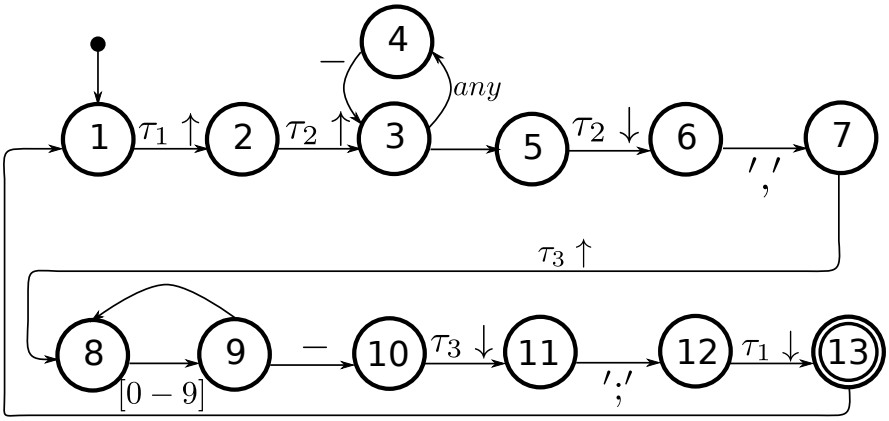


Figure 5.3: Automaton for `((.*?),(\d+);)+` In the diagram, "−" stands for low priority. $\tau_n \uparrow$ is the opening tag for capture group n, likewise, $\tau_1 \downarrow$ is the closing tag for capture group n.

In the NFA, we model greedy repetition or non-greedy repetition of an expression in two steps:

1. We construct an NFA for the expression, without any repetition. Figure 5.3 shows how this plays out in our running example, which contains the expression `.*?`. An automaton for expression `.` is constructed. The expression `.` is modeled as just two nodes labeled 3 and 4, and a transition labeled "any" between them.

2. We add prioritized transitions to model repetition. In our example, repeating is achieved by adding two ε transitions: one from 4 back to 3, to match more than one time any character, and another one from 3 to 4, to enable matching nothing at all. Importantly, the transition from 4 back to 3 is marked as low priority (the "–" sign) while the transition leaving the automaton, from 3 to 5, is unmarked, which means normal priority. This means that the NFA will prefer leaving the repeating expression, rather than staying in it. If the expression were greedy, then we would mark the transition from 3 to 5 as low-priority, and the NFA would prefer to match any character repeatedly.

More generally, the NFA will prefer to follow transitions of normal priority over those of low priority. Rather than formalize this notion of preference on NFAs, we come back to prioritized transitions when discussing the transformation from NFA states to DFA states. For now, note that the NFA we have constructed encodes, in some way, the preference we have for some states over others. The complete rules for NFA construction can be seen in figure 5.4.

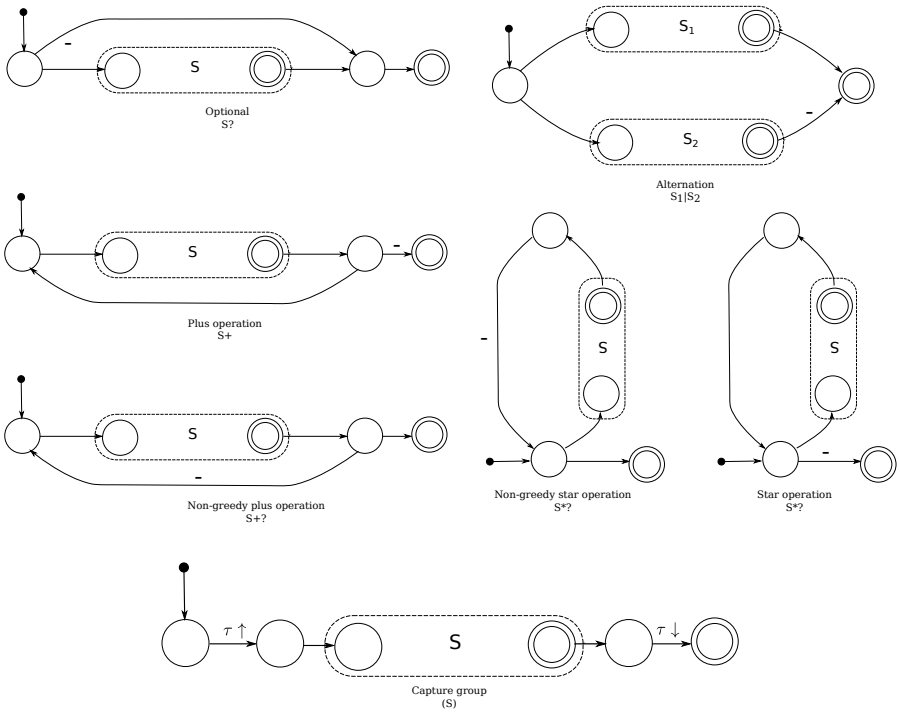


Figure 5.4: Modified Thompson [87] construction of the automaton:Descend into the abstract syntax tree of the regular expression and expand the constructs recursively.

To model capture groups in the NFA, we add *commit tags* to the transition graph. The transition into a capture group is tagged by a commit, the transition to leave a capture group is tagged by another commit. We distinguish opening and closing commits. The NFA keeps track of all times that a transition with an attached commit was used, thus keeping the *history* of each commit. After parsing succeeds, the list of all histories can then be used to reconstruct all matches of all capture groups.

We model histories as linked lists, where the payload of each node is a position. Only the payload of the *head*, the first node, is mutable, the *rest*, all other nodes, are immutable. Because the rests are immutable, they may be shared between histories. This is an application of the Flyweight pattern, which ensures that all of the following instructions on histories can be performed in constant time. Here, the *position* is the current position of the matcher.

5.2.2 DFAs

Our above definition of regular expression assumes a machine that guesses the correct transition through magic. To implement regular expression matching without supernatural intervention, we lazily transform the NFA to a DFA.

A useful metaphor for regular expression matching is that of threads [19]. Whenever we aren't sure which transition to take, we "fork" a thread for every option that we have. This way, when the input is over, there must be at least one thread that guessed correctly at all times. We use the word "thread" here to guide intuition only. Our approach is not parallel.

The key insight is that we keep all "threads" in lock-step. To achieve this, we must be very specific about what constitutes the state of a thread. Since every thread effectively simulates a different NFA state, a thread contains exactly two items: the NFA state it simulates and the history for every tag. Whenever an input character is read, we could iterate over all threads, kill the ones that have no legal transition for the input character, and fork more threads as needed. Trouble strikes when we want to fork a thread for an NFA state that is already running. Not only is an explosion of threads bad for performance, it would also lead to ambiguity: if the two threads disagree on the histories, which one is correct?

Notation. We use the following vocabulary.

DFA states are denoted by a capital letter, e.g. Q, and contain multiple threads.

$$Q = [(q_1, (h_1, h_2, h_3, h_4, h_5, h_6)), (q_2, (h_1, h_2, h_3, h_4, h_7, h_8))]$$

for example means that the current DFA state has one thread in NFA state q_1 with histories $(h_1, h_2, h_3, h_4, h_5, h_6)$ and another thread in NFA state q_2 with the histories $(h_1, h_2, h_3, h_4, h_7, h_8)$. Note that histories can be shared across threads if they have the same matches.

Histories are linked lists, where each node stores a position in the input text. The head is mutable, the rest is immutable. Therefore, histories can share any node except their heads. We write $h = [x_1, \ldots, x_m]$ to describe that matches occurred at the positions $x_1, \ldots, x_m$.

Threads are denoted as pairs (q_i, h), where q_i is some NFA state, and $h = (h_1, \ldots, h_{2n})$ is an array of histories, where n is the number of capture groups. Each thread has an array of $2n$ histories. In an array of histories $(h_1, h_2, \ldots h_{2n-1}, h_{2n})$, history h_1 is the history of the openings of the first capture group, $h2$ is the history of the closings of the first capture group, and so on.

Transitions are understood to be between NFA states, $q_1 \to q_2$ means a transition from q_1 to q_2.

Take, for example, the regular expression `(..)+` matching pairs of characters, on the input string "abcd". The history array of the finishing thread is $[h_1 = [0], h_2 = [3], h_3 = [2, 0], h_4 = [3, 1]]$. Histories h_1 and h_2 contain the positions of the entire match: position 0 through 3. Histories h_3 and h_4 contain the positions of all the matches of capture group 1, in reverse. That is: one match from 0 through 1, and another from 2 through 3.

Our engine executes instructions at the end of every interpretation step. There are four kinds of instructions:

$h \leftarrow \mathbf{p}$ Stores the current position into the head of history h.

$h \leftarrow \mathbf{p} + 1$ Stores the position after the current one into the head of history h.

$h' \mapsto h$ Sets head.next of h to be head.next of h'. This effectively copies the (immutable) rest of h to be the rest of h', also.

$c \uparrow (h)$ Prepends history h with a new node that becomes the new head. This effectively *commits* the old head, which is henceforth considered immutable. $c \uparrow (h)$ describes the opening position of the capture group and is therefore called the opening commit.

$c \downarrow (h)$ This is the same as $c \uparrow (h)$ except that it denotes a closing commit marking the end of the capture group. This distinction is only for clarity, the semantics is the same for opening and closing commits.

Algorithm 1 takes as input a set of threads, an NFA transition graph, and an input character, and returns the set of threads running after the input character has been read. It makes sure that if there could be two threads with the same NFA state, the one that follows greedy matching[1] will survive. At no point of the algorithm are the two states both present in the result and thus in conflict.

In a nutshell, the algorithm works as follows: the threads are racing to capture states. Low priorities are like big hurdles, slowing a thread dramatically. The threads are scheduled, in order, they eat one input character, and then follow only high-priority transitions, saving low-priority transitions on a stack. Once we're out of high-priority transitions, we continue with low priority transitions, until all possible transitions have been followed. Then, we schedule the next input thread. The starting state is obtained from Algorithm 1, taking the starting NFA state as input, with the slight twist that no input character is consumed.

Note that the ordering of threads inside of DFA states is relevant. In Figure 5.3, after reading only one comma as an input, state 7 can be reached from two threads: either from the thread in state 3, via 4, or from the thread in state 6. The two threads are 'racing' to capture state 7. Since in the starting state, the thread of state 6 is listed first, he 'wins the race' for state 7, and 'captures it'. Thus, the new thread of state 7 is a fork of the thread of state 6, not 3. This matters, since 6 and 3 may disagree about their histories.

Example 5.2.1. Execution of algorithm 1: Consider the automaton in figure 5.3 is in the DFA state[2]

$$Q = [(q_6, H_2 = (h_1, h_2, h_7, h_8, h_5, h_6)),\\ (q_3, H_1 = (h_1, h_2, h_3, h_4, h_5, h_6))]$$

This is the case after initialization or before any commas are read.

[1] Non-greedy operators work the same way, just with reversed priorities

[2] This is the starting state, except for the omission of threads that would die immediately after scheduling because there are no consuming transitions attached to their NFA state.

Input : Graph of transitions for an NFA,
Input character a,
Input position pos,
a list of threads $Q = [(q, [h_1, \ldots, h_n])]$, where q is an NFA and $[h_1, \ldots, h_n]$ is an array of histories.
Output: Set of threads R.

```
1  begin
2      R ← [ ]
3      Initialize empty stack buffer
4      Initialize empty stack high
5      Initialize stack low so that (q, h) ∈ Q are retrieved front to back of Q.
6      Mark all threads in low as hungry.
7      // Follow transitions greedily
8      while high and low are not both empty do
9          if high not empty then
10             pop (q', [h_1, ..., h_n]) from high
11         else
12             pop (q', [h_1, ..., h_n]) from low
13             while buffer is not empty do pop (q, [h_1, ..., h_n]) from buffer and add it to R
14
15         end
16         if current thread is marked hungry then
17             foreach a-consuming transition e = q' → q'' do
18                 push (q'', [h_1, ..., h_n]) to high.
19             end
20             jump to top of while.
21         end
22         if q' is marked as seen then jump to top of while loop
23
24         mark q' as seen
25         add (q', [h_1, ..., h_n]) to buffer
26         foreach ε transitions t from q' to q'' do
27             if q'' is marked as seen then continue for loop
28
29             if t is tagged with an open or close tag then
30                 Choose i such that h_i is the history of t's open tag
31                 Make a new history h'
32                 h_i ↦ h'                                          // Copy old history
33                 if t has a open tag then
34                     Let newHistories be [..., h_{i-1}, h', h_{i+1}, ...]
35                     h' ← pos + 1                    // Store position after current
36                 else
37                     // t has a close tag
38                     Choose i' such that h_{i'} is the history of t's open tag
39                     Make a new history h''
40                     h'_{i'} ↦ h''                                 // Copy old history
41                     h'' ← pos                          // Store current position
42                     Commit h'
43                     Commit h''
44                     Let newHistories be [..., h_{i-1}, h', ..., h'', h_{i'+1}, ...]
45                 end
46             end
47             // Push according to priority of transition:
48             if t has low priority then
49                 push (q'', newHistories) to low
50             else
51                 push (q'', newHistories) to high
52             end
53         end
54     end
55 end
```

Algorithm 1: onestep(NFA, a, pos, Q): Compute the follow-up state for DFA state Q.

The algorithm uses two stacks, *high* and *low*. To find the next transition to follow, we pop one from the *high* stack, and only if it is empty do we pop from the *low* stack.

We will pretend for clarity that instructions are executed directly after they are encountered. The actual algorithm collects them and executes them after the *oneStep* call to allow further optimizations.

Furthermore, in the scope of this algorithm, threads have one extra bit of information attached to them: they can be hungry or fed. Hungry threads can only follow transitions that consume characters, fed threads can only follow ε transitions.

This is the execution of *oneStep*(*NFA*, ",", 1, Q):

1. Fill the *low* stack with hungry threads of all states of Q. Now, *low* = $[(q_6, H_2), (q_3, H_1))]$, where the first element is the head of the stack.
2. Initialize *buffer* as an empty stack. The *buffer* stack exists because while following high priority transitions, states are discovered in an order that is reversed with respect to the order in which we would like to output them.
3. Initialize $R = [\,]$, the DFA state under construction.
4. Thread (q_6, H_2) is popped from the *low* stack, since *high* is empty. It is hungry.
5. We iterate all available transitions in the NFA transition graph, and find only $q_6 \to q_7$, which can consume character ",".
6. (q_7, H_2) is pushed to *high* as fed, and we continue the main loop.
7. (q_7, H_2) is taken from the *high* stack. It is fed.
8. (q_7, H_2) is pushed on *buffer*.
9. Since (q_7, H_2) is fed, it follows ε transitions.
10. The available transition $q_7 \to q_8$ is evaluated:
    (a) This transition has a opening tag for capture group 3 on it, and so we'd like to change h_5, the relevant history (see definition of Q above). However, since we're *spawning* a new thread, we cannot change h_5 itself. Instead, we copy h_5, and change the copy.
    (b) A new history h is created.
    (c) $h_5 \mapsto h$. Note that this is constant time, no matter how many entries h_5 already has.
    (d) $h \leftarrow \mathbf{p} + 1$. This is the position after the ",", because the comma was eaten before the capture group starts.
    (e) Create $H_3 = h_1, h_2, h_7, h_8, h, h_6)$ as a copy of H_2, with h in the appropriate position.
    (f) (q_8, H_3) is pushed on the *high* stack.
11. (q_8, H_3) is taken from the *high* stack.
12. It is pushed on *buffer*. *buffer*= $[(q_8, H_3), (q_7, H_2)]$

13. It can follow no further transitions and dies.
14. We discover that the *high* stack is empty.
15. We now flush *buffer*: $R = [(q_8, H_3), (q_7, H_2)]$, *buffer*= [ ]. Note that now, R contains two threads in the reverse order in which they were discovered.
16. (q_3, H_1) is popped from *low*. It is hungry.
17. (q_3, H_1) is one of the two threads that constitute the input of this algorithm. Note how the other, (q_6, H_2), got a chance to follow all of its transitions before (q_3, H_1) was first popped off the low stack.
18. The only transition that consumes "," is $q_3 \to q_4$:
    (a) (q_4, H_1) is pushed to *high* as a fed thread.
19. (q_4, H_1) is popped from *high*
20. It is pushed to the *buffer*. *buffer*= (q_4, H_1)
21. $q_4 \to q_3$ is visited.
    (a) Thread (q_3, H_1) is pushed to low = (q_3, H_1), because $q_4 \to q_3$ has low priority.
22. We flush the *buffer* again: $R = [(q_8, H_3), (q_7, H_2), (q_4, H_1)]$ Note how (q_4, H_1) appears *last* in R.
23. (q_3, H_1) is taken from the *low* stack, because *high* is empty.
24. It is added to *buffer*.
25. $q_3 \to q_5$ is visited:
    (a) (q_5, H_1) is pushed to *high*.
26. (q_5, H_1) is taken from the *high* stack.
27. It is added to *buffer*.
28. $q_5 \to q_6$ is visited and contains the closing commit of the second capture group:
    (a) Two histories are created to store the new positions of both the start and the end of the capture group. This ensures that other threads will not corrupt the memory.
    (b) A new history h is for the opening of the capture group.
    (c) A new history h' is created for the closing position.
    (d) $h_3 \mapsto h$. See the definition of Q above, to see that $h3$ is the opening capture group position of H_1.
    (e) $h_4 \mapsto h'$.
    (f) $h' \leftarrow \mathbf{p}$. This is the position of the ",".
    (g) Create a new history array, with h and h' in place. $H_4 = [h_1, h_2, h, h', h_5, h_6]$

(h) (q_6, H_4) is pushed to *high*.

29. (q_6, H_4) is taken from the *high* stack.
30. It is added to the *buffer*.
31. Both stacks are empty:
32. We flush our *buffer*:

$$R = [(q_8, H_3), (q_7, H_2), (q_4, H_1), \\ (q_6, H_4), (q_5, H_1), (q_3, H_1)]$$

33. R is returned.

The output contains six threads, but three of them, (q_7, H_2), (q_4, H_1), (q_5, H_1), will die as soon as they are scheduled in the next iteration of the algorithm, because there are no outgoing non-ε transitions attached to their NFA states.

The overall run time of algorithm 1 is $O(m \log(m))$. This follows from the fact that there can be only one thread per NFA state, and therefore there are only m threads. To achieve this bound, we must be able to copy arrays of histories in time $O(\log(m))$. This is easily achieved replacing the array by a persistent [27] data structure to hold the histories. A persistent treap, sorted by array index, has all necessary properties[3].

5.3 Implementation

While repeatedly calling algorithm 1 would be sufficient to reach the theoretical time bound we claimed, practical performance can be dramatically improved by avoiding to construct new states. Instead, we build a *transition table* that maps from old DFA states and an input range to a new DFA state, and the instructions to execute when using the transition. We build the transition table, including instructions, as we go. This is what we mean when we say that the DFA is *lazily compiled*.

5.3.1 DFA transition table

The DFA transition table is different from the NFA transition table, in that the NFA transition table contains ε transitions and may have more than one transition from one state to another, for the same input range. DFA transition tables allow no ambiguity.

Our transition tables, both for NFAs and DFAs, assume a transition to map a consecutive range of characters. If, instead, we used individual characters, the table size would quickly become unwieldy. However, input ranges can quickly become confusing if they are allowed to intersect. To avoid this, and simplify the code dramatically, while keeping the transition table small, we use the following trick. When the regular expression is parsed, we keep track of all input ranges that occur in it. Then, we split them until no two input ranges intersect.

[3] Clojure [37] features a slightly more complex data structure under the name of 'persistent vectors'. Jean Niklas L'orange offers a good explanation in "Understanding Clojure's Persistent Vectors", `http://hypirion.com/musings/understanding-persistent-vector-pt-1`.

After this step, input ranges are never created again. Doing this step early in the pipeline yields the following invariant: it is impossible to ever come across intersecting input ranges.

To give us a chance to ever be in a state that is already in the transition table, we check, after executing algorithm 1, `oneStep`, whether there is a known DFA state that is *mappable* to the output of `oneStep`. If `oneStep` produced a DFA state Q, and there is a DFA state Q' that contains the same NFA states, in the same order then Q and Q' may be mappable. If they are, then there is a set of instructions that move the histories from Q into Q' such that, afterwards, Q' behaves precisely as Q would have. Algorithm 2 shows how we can find a mappable state, and the needed instructions. The run time of Algorithm 2 is $O(m)$, where m is the size of the input NFA.

```
   Input  : Q = [(q_i, h_i)]_{i=1...n} is a DFA state.
   Output: A state Q' that Q is mappable to.
           The ordered instructions m that reorder the memory locations of Q to Q' and don't
   interfere with each other.
 1 begin
 2     foreach Q' that contains the same NFA states as Q, in the same order do
 3         /* Invariant: For each history H there is at most one H'                */
 4         /* so that H ← H' is part of the mapping.                               */
 5         Initialize empty bimap m
 6         /* A bimap is a bijective map.                                          */
 7         foreach q_i = q'_i with histories H and H' respectively do
 8             for i = 0 ... length(H) − 1 do
 9                 if H(i) is in m as a key already and does not map to H'(i) then
10                     Fail
11                 else
12                     /* Hypothesize that this is part of a valid map            */
13                     Add H(i) ↦ H'(i) to m;
14                 end
15             end
16         end
17     end
18     /* The mapping was found and is in m.                                       */
19     sort m in reverse topological order so that no values are overwritten.
20     return Q' and m
21 end
```

Algorithm 2: $findMapping(Q)$: Finding a state that Q is mappable to in order to keep the number of states created bound by the length of the regular expression.

5.3.2 DFA execution

With these ingredients in place, the entire matching algorithm is straightforward. In a nutshell, we see if the current input appears in the transition table. Otherwise, we run `oneStep`. If the resulting state is mappable, we map. More formally, we can see this in algorithm 3. Here, algorithm 3 assumes that algorithm 1 does not immediately execute its instructions, but returns them back to the interpreter, both for execution and to feed into the transition table.

5.3.3 Compactification

The most important implementation detail, which brought a factor 10 improvement in performance, was the use of a compactified representation of DFA transition tables whenever possible. Compactified, here, means to store the transition

```
   Input  : input is a sequence of characters.
   Output: A tree of matching capture groups.
 1 begin
 2     // Lazily compiles a DFA while matching.
 3     Set Q to startState.
 4     // A thread is an NFA state, with an array of histories.
 5     Let Q be all threads that are reachable in the NFA transition graph by following ε transitions
       only.
 6     Execute instructions described in algorithm oneStep, when walking ε transitions.
 7     // Create the transition map of the DFA.
 8     Set T to an empty map from state and input to new state and instructions.
 9     // Consume string
10     foreach position pos in input do
11         Let a be the character at position pos in input.
12         if T has an entry for Q and a then
13             // Let the DFA handle a
14             Read the instructions and new state Q' out of T
15             execute the instructions
16             Q ← Q'
17             jump back to start of for loop.
18         else
19             // lazily compile another DFA state.
20             Run oneStep(Q, a) to find new state Q' and instructions
21             Run findMapping(Q', T) to see if Q' can be mapped to an existing state Q''
22             if Q'' was found then
23                 Append the mapping instructions from findMapping to the instructions found
                   by oneStep
24                 Execute the instructions.
25                 Add an entry to T, from current state Q and a, to new state Q'' and
                   instructions.
26                 Set Q to Q''
27             else
28                 Execute the instructions found by oneStep.
29                 Add an entry to T, from current state Q and a, to new state Q' and
                   instructions.
30                 Set Q to Q'.
31             end
32         end
33     end
34 end
```

Algorithm 3: interpret(input): Interpretation and lazy compilation of the NFA.

table as a struct of arrays, rather than as an array of structs, as recommended by the Intel optimization handbook [18, section 6.5.1]. The transition table is a map from source state and input range to target state and instructions. Following Intel's recommendation, we store it as an object of five arrays: `int[] oldStates, char[] froms, char[] tos, Instruction[][] instructions, int[] newStates`, all of the same length, such that the ith entry in the table maps from `oldStates[i]`, for a character greater than from[i], but smaller than `to[i]`, to `newStates[i]`, by executing `instructions[i]`. To read a character, the engine now searches in the transition table, using binary search, for the current state and the current input character, executes the instructions it finds, and transitions to the new state.

However, the above structure isn't a great fit with lazy compilation, as new transitions might have to be added into the middle of the table at any time. Another problem is that, above, the state is represented as an integer. However, as described in the algorithm, a DFA state is really a list of threads. If we need to lazily compile another DFA state, all of the threads need to be examined.

The compromise we found is the following: The canonical representation of the transition table is a red-black tree of transitions, each transition containing

n	13	14	15	16	17	18	19	20
Oracle	241	484	1003	1874	3555	7381	14561	30116
Ours	225	252	273	32	327	352	400	421

Table 5.1: Matching times, in microseconds, for matching $a?^n a^n$ against input a^n.

source and target DFA state (both as the full list of their NFA states, and histories), an input range, and a list of instructions. This structure allows for quick inserting of new DFA states once they are lazily compiled. At the same time, lookups in a red-black tree are logarithmic. Then, whenever we read a fixed number of input characters without lazily compiling, we transform the transition table to the struct of arrays described above, and switch to using it as our new transition table. If, however, we read a character for which there is no transition, we need to de-optimize, throw away the compactified representation, generate the missing DFA state, and add it to the red-black tree.

The above algorithm chimes well with the observation that usually, regular expression matching needs only a handful of DFA states, and thus, compactifying can be done early, and only seldom needs to be undone.

5.3.4 Intertwining of the pipeline stages

The lazy compilation of the DFA when matching a string enables us to avoid compiling states of it that might never be necessary. This allows us to avoid the full power set construction [83], which has time complexity of $O(2^m)$, where m is the size of the NFA.

5.4 Benchmark

All benchmarks were obtained using Google's caliper[4], which takes care of the most obvious benchmarking blunders. It runs a warm-up before measuring, runs all experiments in separate VMs, helps circumvent dead-code detection by accepting the output of dummy variables as input, and fails if compilation occurs during experiment evaluation. The source code of all benchmarks is available, together with the sources of the project, on Github. We ran all benchmarks on a 2.3 GHz, i7 Macbook Pro.

As we will see in Section §5.5, there is a surprising dearth of regular expression engines that can extract nested capture groups — never mind extracting entire parse trees — that do not backtrack. Back-tracking implementations are exponential in their run-time, and so we see in Figure 5.5 (note the log plot) how the run-time of "java.util.regex" quickly explodes exponentially, even for tiny input, for a pathological regular expression, while our approach slows down only linearly. The raw data is seen in Table 5.1.

In the opposite case, in the case of a regular expression that's crafted to prevent any back-tracking, java.util.regex outperforms our approach by more than factor 2, as seen in Table 5.2 – but bear in mind that java.util.regex does not extract parse trees, but only the last match of all capture groups. A backtracking

[4] `https://code.google.com/p/caliper/`

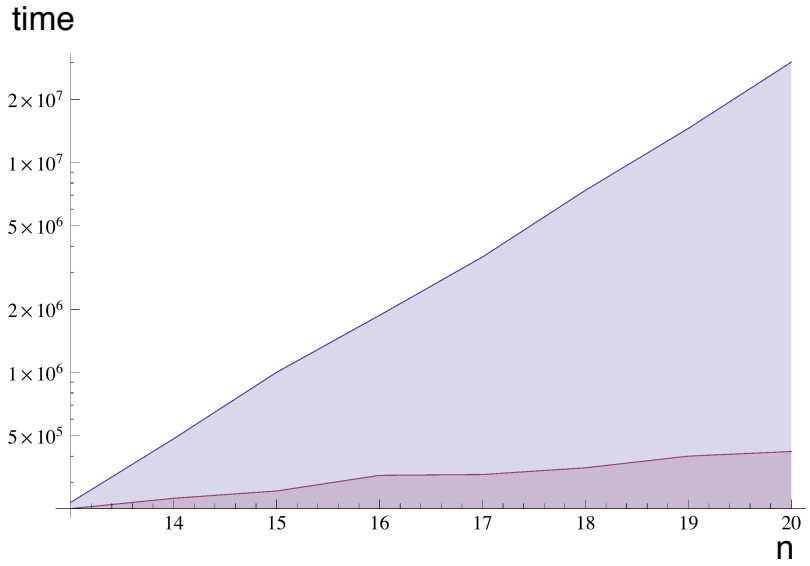


Figure 5.5: Time in nanoseconds for matching $a?^n a^n$ against input a^n. Bottom (purple) line is our approach, top (blue) line is java.util.regex.

Tool	time
JParsec	4,498
java.util.regex	1,992
Ours	5,332

Table 5.2: Matching regular expression `((a+b)+c)+` against input (aˆ200bc)ˆ2000, where aˆ200 denotes 200 times character 'a'. Time in microseconds.

implementation that actually does produce complete parse trees is JParsec[5], which, as seen in Table 5.2, performs on par with our approach for a regular expression that is hand-crafted to prevent back-tracking.

Since JParsec is back-tracking, its worst case performance is exponential, though it could be improved, at the cost of much memory, through memoization, to perform in cubic time [31].

Note that because java.util.regex achieves its back-tracking through recursion, we had to set the JVM's stack size to one Gigabyte for it to parse the input. Since default stack size is only a few megabytes, this makes using java.util.regex a security risk, even for unproblematic regular expressions that cannot cause backtracking, since an attacker can potentially force the VM to run out of stack space.

A more realistic example, neither chosen to favor back-tracking nor to avoid it, extracts all class names, with their package names, from the project sources itself. As seen in Table 5.3, our approach outperforms java.util.regex by 40%,

[5] `http://jparsec.codehaus.org`

Tool	time
java.util.regex	11,319
Ours	8,047

Table 5.3: Runtimes, in microseconds, for finding all Java class names in all .java files in the project itself. The regular expression used is `(.*?([a-z]+\.)*([A-Z][a-zA-Z]*))*.*?`.

even though our approach constructs the entire parse tree, and thus all class names, while java.util.regex outputs only the last matched class name. We omit JParsec, since it offers no direct way of expressing non-greedy matching.

5.5 Related work

Our algorithm is a modification of Laurikari's algorithm [55], which is itself a modified power set construction algorithm [83, p. 55].

While there is no shortage of books discussing the usage of regular expressions, the implementation side of regular expression has not been so lucky. Cox is spot-on when he argues that innovations have repeatedly been ignored and later reinvented [19, 20, 21].

This chapter is no exception. The author had set out to implement Laurikari's TDFA algorithm [55], only to discover that Laurikari's description of a TDFA is so far from complete that it can rightfully only be called the sketch for an algorithm. Only late in the process did we discover that the blanks had already been filled by Kuklewicz in the course of his implementation of TDFAs in Haskell [54]. Kuklewicz enshrined his added insight into Haskell library, but never published the algorithm as a whole. If the history of regular expressions is evidence of one thing, it is that source code is a terrible medium to convey algorithms.

The situation dramatically improved with Cox's simple and concise explanation of regular expression matching [19]. It seems ironic that this well-versed author published this influential work on his website. The joke, however, may be on Academia's side.

When the practitioners acknowledge each other's work, we can't help but disagree almost universally with the characterizations they produce. Sulzmann and Lu [85] call Kuklewicz's work an "implementation" of Laurikari's algorithm, although Laurikari's algorithm is far too incomplete for that statement to be fair. Laurikari's algorithm is referred to as a POSIX-style automaton. In truth, Laurikari leaves the matching strategy entirely open. It was Kuklewicz that found out how to get POSIX-style matching out of Laurikari's TDFA.

Cox says that Laurikari's TDFA is a reinvention of Pike's algorithm, used in the text editor sam [67]. While sam was released as free software, no description of his regular expression matching, besides the source code, was published. Sam uses Thompson's NFA, but adds submatch tracking. This seems unfair in that Laurikari's allows for far more aggressive reuse of old states than Thompson allows. This should lead to Laurikari's TDFA having fewer states, and therefore

better performance, than even Google's RE2[6], which uses Pike's algorithm. This is not confirmed by the benchmarks by Sulzmann and Lu [85], but they offer an explanation: in their profiling, they see that all Haskell implementations spend considerable time decoding the input strings. In other words, the measured performance is more of an artifact of the programming environment used.

Another mistake that permeates the scarce literature is to call regular expression matching linear. As Sedgewick points out correctly [80], Thompson's NFA matching is of complexity $O(mn)$, where m is the size of the input NFA, and n is the size of the input string. To call this linear means to assume that m is fixed, which is not justified. It may well be true that, at present, m tends to be small. But that is a natural consequence of the algorithms not scaling very well with m. If they did, that would allow for fast feature extracting from text[7]. Therefore, in this chapter, we consider the state of the art algorithms to be quadratic, since both m and n are part of the input to a regular expression matcher. We cannot rule out that a linear algorithm exists, in fact, we hope for it. To insist that regular expression matching is done in linear time is to insist that the optimal algorithm has already been found; that is probably not true.

Sulzmann and Lu add to the table a new matching strategy that yields good practical performance, although the theoretical bounds are considerably worse than the state of the art, at $O(n^2m)$ [85].

Kearns [47] describes the first linear-time algorithm for RE parsing. Dubé and Feeley [28] produce parse trees in linear time using lists as their data representation. Nielsen and Henglein [63] improve on it by storing the trees in bit-coded form. Neither produces a greedy or non-greedy parse. Grathwohl [33] produce a greedy parse in two over the input. In comparison, our approach allows full control over greediness, even for subexpressions, thus generalizing over all previous approaches. It runs in only one pass and supports character ranges, lazy compilation, and compactified DFA representation for practical performance.

5.5.1 Motivation

Not only clone detectors benefit from lightweight parsing. Regular expressions make for scaleable and efficient lightweight parsers.[46] The first step of processing big data is often to parse strings. As an example, consider log files. As Jacobs[42] noted, "What makes most big data big is repeated observations over time and/ or space," and thus log files grow large frequently. At the same time, they provide important insight into the process that they are logging, so their parsing and understanding is important.

The parsing abilities of regular expression have provoked Meiners to declare that for intrusion detection, "fast and scaleable RE matching is now a core network security issue." [62]

For example, Arasu et al. [2] demonstrate how regular expressions are used in Bing to validate data, by checking whether the names of digital cameras in their database are valid.

[6] `https://code.google.com/p/re2/`

[7] To check if a document contains features $f_1, f_2, \ldots, f_n$, we would match the document against regular expression `(`f_1`)|(`f_2`)|...|(`f_n`)`.

5.6 Conclusion

Regular expressions can be used to robustly approximate full parsing of source code in linear time. Our approach can produce entire parse trees while matching regular expressions. The performance is on par with traditional back-tracking solutions if no backtracking ever happens, exponentially outperforms back-tracking approaches for pathological input, and in a realistic scenario outperforms backtracking by 40%, even though our approach produces the full parse tree, and the backtracking implementation doesn't. All source code and all benchmarks are available under a free license on Github[8].

The fact that our approach is linear in the text input means that it scales well. Linearity is a key ingredient to scaleability. Every algorithm that isn't linear requires to overcompensate load growth by system growth. This recommends lightweight parsing for use in our clone detector.

Acknowledgements

This chapter is based on work I did with Aaron Karper and Oscar Nierstrasz [75]. This work would not have been possible without Aaron. Aaron and I spent long afternoons drawing automata on blackboards, until everything worked out.

While much related work has been done, I am ashamed to admit our starting point was practically only Laurikari's sketch of an algorithm [55]. After our own approach was all set and done, I was surprised by the wealth of literature on the issue. But moreover, I was immensely proud of my collaboration with Aaron. In all our ignorance, we still pushed the envelope of what can be done.

Our collaboration was productive, friendly, but most of all, great fun.

[8] `https://github.com/nes1983/tree-regex`

Chapter 6

Cells

In this chapter, we explain MapReduce pipelines, the last cornerstone for a scaleable clone detector. MapReduce [25] is a programming paradigm to distribute and parallelize computations. A *MapReduce pipeline* is a library that allows running several MapReduce programs in sequence. While a simple shell script that runs MapReduce programs in sequence suffices in theory, we suggest a different approach, where the developer specifies a sequence of mappers, which are then automatically composed into MapReduce programs. This simplifies the writing of scaleable software, and makes it more debuggable. En passant, we will explain MapReduce and Bigtable, and thus show that expressing a computation as a Cells program makes it *scaleable* – if asymptotic run time permits it.

Our MapReduce pipeline is named *Cells*. It lets developers express pipelines of MapReduce jobs, so that the same pipeline can be executed either in parallel on one machine, or in a MapReduce cluster.

The code examples in this chapter give away some of the working of our clone detector. The full description will follow in chapter 7.

In MapReduce, the developer specifies two operations: map and reduce. On a cluster with distributed storage, they are used as follows. The map operation reads the input data, and processes it. The output of the map operation is grouped and sorted, and passed, in a distributed but non-persistent way, to the reduce operation. The output of the reduce operation is written back into the cluster. Map and reduce are different: map can read any format, but must write key-value pairs. Conversely, reduce can write any format, but must read groups of key-value pairs.

Cells is non-invasive in that code that uses Cells is in the style of ordinary Java. It features no special looping constructs.

In Cells, all objects are encoded into cells, a binary format of row header, column header, and cell contents. Just like Bigtable cells, there is an implicit grouping and ordering: cells with the same row header are grouped into the same row, and within rows, cells are sorted by their column headers. This simple idea is powerful enough to express pipelines of MapReduce jobs without distinguishing between map and reduce. Furthermore, it lets us express computations without distinguishing between reading from Bigtables, file directories, or lists.

The resulting computational model is efficient, surpassing PLINQ for non-distributed computations.

There exist a number of frameworks that help developers express a com-

putation in non-blocking terms, so that the framework can then parallelize the program on behalf of the developer. However, they tend to be invasive, by which we mean that programs written in those frameworks look fundamentally different from those written without it. As an example, consider FlumeJava [13] and PLINQ [61, 38], frameworks to express parallel computations in Java and C#, respectively.

The following is a valid statement in PLINQ:

```
from c in Customers.AsParallel()
where c.Address.City = "Seattle"
select c.Name, c.Phone
```

The following is an example in FlumeJava:

```
PTable<URL,DocInfo> backlinks =
  docInfos.parallelDo(new DoFn<DocInfo,Pair<URL,DocInfo>>() {
    void process(DocInfo docInfoEmitFn<Pair<URL,DocInfo>> emitFn) {
      for (URL targetUrl : docInfo.getLinks()) {
        emitFn.emit(Pair.of(targetUrl, docInfo));
      }
    }
  }, tableOf(recordsOf(URL.class),
  recordsOf(DocInfo.class)));
PTable<URL,Collection<DocInfo>> referringDocInfos =
backlinks.groupByKey()
```

Both languages play the same trick: they bolt a functional language on top of their host language, and then try to optimize the functional constructs into parallel-executable chunks. In the case of FlumeJava, the developer is supposed to chain together many `parallelDo` invocations, after which FlumeJava will execute them in a distributed MapReduce using as few MapReduce stages as possible. To that effect, where ordinary Java code would loop to select a subset, FlumeJava demands functional equivalents, provided by the library, effectively making FlumeJava a programming language in its own right. Even collections are largely replaced by their parallel counterparts, `PCollection`s. It should be obvious to anybody versed in C# or Java that the above programs don't much resemble ordinary programs in the host language. The framework dominates the vocabulary and format of the program.

In contrast, Cells[1] was conceived from the viewpoint that well-written software, even if it is not meant to be run in parallel, is typically expressed in a pipeline fashion. That is, the computation is broken up into individual stages that are chained together from the outside. Therefore, a Cells program should look very similar to any normal program written in a pipeline fashion.

The following snippet and all following snippets are taken from a clone detector that uses Cells. They are simplified, but only slightly.

The following is a *mapper* — a stage in a pipeline — that extracts snippets of 5 lines from Java source code after the input source code has been broken up into functions.

[1] All code is available on Github. `https://github.com/nes1983/cc`

```
class SnippetExtractor implements Mapper<String, Snippet> {
  public void map(Function first, OneShotIterable<Function> functions, Sink<Snippet> sink)
     {
    for (Function fun : functions) {
      for (int frameStart = 0; frameStart + 5 <= fun.nLines(); frameStart++) {
        byte[] hash = badHash(getLines(fun.getString(), frameStart, 5));

        Snippet snip = Protos.Snippet.newBuilder()
            .setFunction(fun)
            .setHash(ByteString.copyFrom(hash))
            .build();
        sink.write(snip);
      }
    }
  }
}
```

Unlike the PLINQ or FlumeJava examples, the vocabulary of this snippet is not primarily about being parallel or functional, or even about being part of a MapReduce job; it is about extracting lines from functions. It runs a sliding window of 5 lines over a function to compute a bad hash for each window. Hash value and function together form a snippet `Snippet`, which is written out.

Only a few unusual features stand out: the output is written into a Sink instead of being returned, the input collection is a OneShotIterable, and there is an unused parameter `first`. We'll discuss these features in more detail.

To run the entire clone detector, we feed all stages into a *Pipeline*. There are pipelines for local execution, and pipelines for distributed (in Hadoop[2]) execution. However, Cells was designed to allow the mappers to be written in way that is agnostic about whether they're running locally, or distributed in a cluster. This is how a pipeline can be executed, remotely or locally, depending on the `pipe` argument.

```
void run(Pipeline<Repo, Clone> pipe) {
  pipe
    .influx(new RepoCodec())
    .map(new SnippetExtractor())
    .shuffle(new SnippetCodec())
    .map(new RoughCloner())
    .shuffle(new CloneCodec())
    .mapAndEfflux(new FineCloner(), new CloneCodec())
}
```

The above snippet runs a pipeline of three stages: `SnippetExtractor`, `RoughCloner` and `FineCloner`. Since we have three stages to execute, but MapReduce only has two phases, our library will have to break this into two MapReduce jobs that are executed in succession. The necessary grouping and sorting is expressed through the codecs between the mappers.

In section §6.1, we present the gist of how cells is meant to be used. In section §6.2, we explain the design rationale. In section §6.3, we give the most interesting implementation details. In section §6.4, we show a set of benchmarks that compare cells to PLINQ and plain Hadoop. In section §6.5, we list related work. In section §6.6, we draw our conclusions.

6.1 Cells in a nutshell

Cells is a library that parallelizes the execution of jobs that are written in the style of a chain of MapReduce [25] jobs. In this style, all computation must be

[2]Hadoop is Apache's MapReduce implementation. `http://hadoop.apache.org`

expressed as a pipeline of the operations map (filter), group, and sort. The basic building block of the Cells library is a cell. Cells are modeled after Bigtable [14] cells: they have a *row key*, a *column key*, and cell contents for the payload of the cell. While the row key and column key must each be non-empty byte arrays, the cell contents may be an empty byte array. Cells are implicitly *grouped* by row key into rows, and *ordered* by column key within rows. Two cells with identical row and column key are assumed to be identical, and therefore one is silently suppressed. Row and column keys are assumed to be short (thousands of bytes would be big), but there is no assumption on the size of a cell's contents.

Together, cells form tables, very similar to Bigtables. Bigtables, and therefore our tables, have a few unusual properties. There's no limit on the number of columns that a row can have. They can be sparse, meaning that rows don't need to share column keys. It's idiomatic to store pieces of data, such as a timestamp, in the column header, leading to potentially long rows. It cannot even safely be assumed that rows fit into memory. However, it is just as idiomatic to have just a single column key shared across all rows, effectively degenerating the table to a set of key-value pairs. There's no limit on the number of rows.

This explains the unusual features we saw in Mappers earlier: they accept their input as `OneShotIterable`s, `Iterable`s that can only be iterated once, and write their output into a sink, rather than returning it as a collection. Together, they allow processing more data than fits into RAM, by reading input from and spilling output to disk as necessary.

Our design follows the following principles:

- All input to and output from computations is encoded using cells.
- All input and output is grouped into rows, so that cells with same row key are in the same row. The elements inside the row are sorted by column key.
- All computation runs within mappers. Mappers are each single-threaded, but many mappers can be instantiated to run in parallel. A mapper processes one row at a time, in its entirety. Rows are never shared between mappers.

In sum, these principals ensure that reading and writing to a mapper is always equivalent to reading its input from, our writing its output to a Bigtable. We'll see how this uniformity helps debugging.

We'll demonstrate Cells' features through the design of a clone detector that scales to detecting clones across all Java projects, in all versions available on the internet.

6.1.1 Codecs and cells

Above, we saw an example of a mapper that runs a sliding window of 5 lines over it to extract text snippets. Let us examine how the sorting and grouping of cells is used by examining the output of the mapper, all snippets of 5 lines. As we saw above, they are written into a sink. The sink then uses a *codec* to encode the snippet into cells, and stores the cells into a table.

Codecs decide the *cell layout*, *i.e.,* which part of our data goes into the row key, the column key, or into the cell contents. Expressing grouping and sorting

in terms of Bigtable cells is necessary if we want to store the intermediate result between two MapReduce jobs in a Bigtable. As we will see later, lifting cells to a basic building block makes the distinction between Map and Reduce disappear.

Choosing the row and column key is usually straightforward: if a specific sorting is needed, that decides the column key. If a specific grouping is needed, that decides the row key. Otherwise, one needs to be on guard about collisions: if several cells of identical row and column key are sent to a sink, only one will be stored.

Therefore a codec is not merely a way of serializing data. Codecs specify the cell layout, and thus the grouping and sorting of our data. However, the right layout to pick will depend on what we intend to do in the next stage of the pipeline. In the case of our clone detector, will only forward snippets that occur in at least two functions, filtering out everything that isn't a clone. This means that we want to group snippets together with the location where we found them by their snippet hash. This is enough information to specify the codec. To achieve grouping by snippet hash, the snippet hash is encoded to be the row key. While the sorting inside the row is irrelevant for the next mapper, we still have to choose a column key that avoids collisions between snippets from different functions. Picking the origin of a snippet, its function, as the column key avoids all collisions. The resulting codec is as follows.

```
class SnippetCodec implements Codec<Snippet> {
  @Override
  public Cell<Snippet> encode(Snippet s) {
    return Cell.make(s.getHash(), s.getFunction(), s.toByteString());
  }

  @Override
  public Snippet decode(Cell<Snippet> encoded) {
    return Snippet.parseFrom(encoded.getCellContents());
  }
}
```

This codec duplicates the snippet hash and function hash: they are stored in row and column key, respectively, and again in the cell contents. This pattern simplifies the decode operations at the cost of storage space.

Since in the MapReduce paradigm, all computations must be expressed as a combination of mapping, grouping and sorting, the design of the codec is not just a detail. Instead, codecs are a fundamental part of a pipeline. This also explains why Cells ships without any pre-defined Codecs. Codecs specify the grouping and sorting of data, and should therefore be specific to their pipeline.

While this puts some constraints on how computations must be expressed, the constraints are close to how any program must be structured that runs in a pipeline fashion: computation is broken into named stages that are chained together.

6.1.2 Lookup tables and side outputs and inputs

Besides writing output to the sink, Cells brings some support for *side outputs* and *side inputs*, *i.e.,* writing to and reading from other tables.

The effect of `SnippetCodec` was to group Snippets by hash. Therefore, in the next stage of the pipeline all snippets that are similar to one another now occur in the same row. The next pipeline stage now filters out snippets that weren't similar to any other snippets, *i.e.,* rows of size 1. Next, since any two snippets

in a row are similar, and a clone is a pair of two snippets, the clones that can be extracted from a row is the cross product of the row with itself. However, rows can get awfully long. In fact, the cross product of a large row might easily go beyond our computational capabilities. Very long rows arise with very popular snippets, such as constructors, or getters and setters. We can simply discard popular snippets, since we're in the business of detecting interesting clones, whereas ubiquitous clones, such as constructors that share the same shape, are necessarily uninteresting. However, there's a better way: while we don't output popular snippets as part of the pipeline stage, we set them aside into a popular snippets table. This will keep them out of the deadly cross product, and prevent them from becoming clones in their own right. However, later, when we stitch together individual clones into bigger ones, they'll still be available since we've set them aside.

The following mapper filters out rows of size ≤ 1, and outputs the cross product, unless it is too large, in which case we write it into a side output.

```
class CloneCrossProduct implements Mapper<Snippet, Clone> {
  @Inject
  @PopularSnippets
  Sink<Snippet> popularSnippets;

  @Override
  public void map(Snippet first, OneShotIterable<Snippet> rowIterable, Sink<Clone> sink)
      throws IOException, InterruptedException {
    // rowIterable is not guaranteed to be iterable more than once, so copy.
    Collection<Snippet> row = ImmutableList.copyOf(rowIterable);

    if (row.size() <= 1) {
      return; // prevent processing non-recurring hashes
    }

    // special handling of popular snippets
    if (row.size() >= POPULAR_SNIPPET_THRESHOLD) {
      for (Snippet loc : row) {
        popularSnippets.write(loc);
      }
      return;
    }

    for (Snippet thisSnip : row) {
      for (Snippet thatSnip : row) {
        if (thisSnip.getFunction().compareTo(thatSnip.getFunction()) < 0) {
          sink.write(Clone.newBuilder().setThisSnippet(thisSnip).setThatSnippet(thatSnip).
     build());
        }
      }
    }
  }
```

To make the injection of `@PopularSnippets Sink<Snippet>` work, we need to teach Cells about the existence of the extra table. Cells offers an abstraction of Bigtables, called a `Table`, and a helper method to get one. Furthermore, Cells ships with an implementation of `Table` that runs wholly in memory. Therefore, we can write Cells jobs locally, without ever talking to a Bigtable.

Cells offers a helper method `installTable`, using Guice for dependency injection[3], to configure which kind of table we would like to have injected into our mapper. The following code configures `PopularSnippets` to be either a Bigtable, or an in-memory table, depending the value of `storageModule`.

[3] `https://code.google.com/p/google-guice/`

```
installTable(
  PopularSnippets.class, // Annotation
  Snippet.class, // Generic type of the table
  PopularSnippetsCodec.class, // Codec used by the table
  storageModule, // HBaseModule or InMemoryModule
  new TableModule<>("PopularSnippets")); // Table name. More settings if needed.
```

Above, we use `installTable` to obtain a Sink for the table, but we could equally well have asked for a Source to read from it, or for a LookupTable for random access into it; `installTable` installs all three.

6.1.3 Pipelines

We can now return to how a pipeline can be obtained. The following snippet obtains a pipeline that reads its input from a list: Here, the method Cells.shard does its best to split the input into even-sized shards of cells.

```
List<Repo> list = ...
CellSource<Repo> = Cells.shard(Cells.encode(list));
Pipeline<Repo, Snippet> pipe = LocalPipeline.from(source);
```

On the other hand, the following snippet obtains a pipeline for distributed execution:

```
Table<Repo> in = TableAdmin.getInstance().tableNamed("repos");
Table<Snippet> out = TableAdmin.getInstance().tableNamed("snippets");
Pipeline<Repo, Snippet> pipe = HadoopPipeline.fromTo(in, out);
```

6.1.4 Post-processing using Sources

Near the end of the clone detection pipeline, we're left with a moderately-sized set of *clones*, that is a set of a pairs of source code locations that are considered to be similar. It would be nice to see the related groups of clones. A clone can be thought of as an edge in the graph of source code locations. In that case, we're interested in all connected components of that graph. Sadly, graph clustering is notoriously hard to do in parallel [74]. It also isn't necessary, since the volume of clones — unlike the volume of all written source code — is perfectly manageable by a single machine.

To run post-processing on the output of a pipeline, `Pipeline` offers the method `lastEfflux()`, which returns a `Source`, which is just an iterable over decoded rows. For the output of the clone detector, we can run the following:

```
Source<Clone> in = pipe.lastEfflux();
for (Iterable<Clone> row : in) {
  for (Clone c : row) {
    hashToClone.put(c.getThisSnippet().getFunction(), c);
    ...
  }
}
```

6.1.5 Counters

Just like MapReduce[25], Cells provides a facility to count occurrences of various events. For example, user code may want to count the local number of words processed or the number of German documents indexed, *etc.*

As with side inputs and outputs, Cells provides a local implementation, as well as a distributed variant, and an abstraction so Mappers don't have to know in which context they're running. One can use a Cells counter as follows.

```
public static class IdentityMapper implements Mapper<Integer, Integer> {
    @GermanDocs
    @Inject
    Counter germanCount;

    @Override
    public void map(Integer first, OneShotIterable<Integer> row, Sink<Integer> sink) {
        try {
          for (Integer i : row) {
            sink.write(i);
          }
        } catch(GermanDetectedException e) {
          germanCount.increment(1L);
        }
      }
    }
  }
```

As with tables, counters need to be installed for the injection to work.

```
installCounter(GermanDocs.class, counterModule);
```

6.2 Design rationale

Cells is designed from the viewpoint that ordinary code, written with no knowledge of MapReduce, is often written in pipeline style anyway. By pipeline style, we mean that there are different stages of computation, that data is passed in a mostly-linear way from one stage to another, and that the individual pieces are tied together from the outside. Cells programs are meant to look like ordinary pipeline code.

In Cells, reading any input, even if it is not read from a Bigtable, is read as if it were stored in a Bigtable. This means that there is always a well-defined grouping and sorting, independent of the data source. This uniformity means that we can always, if needed, store intermediate output in a Bigtable, and resume computation from there.

6.2.1 Debuggable

In classical MapReduce, Mappers and Reducers are different. Mappers can read all kinds of things, but must output key-value pairs. Reducers can write all kinds of things but must read key-value pairs. In Cells, the restriction of all mappers to read and write cells nullifies the distinction. This allows the following.

In MapReduce, re-running a reducer, say, to fix a bug in it, always also requires re-running the mapper. This can be egregious if a mapper takes a long time to run, and the reducer fails almost immediately after being started. Since, in Cells, the computational model is always equivalent to reading from a Bigtable, we can always write intermediate output to a Bigtable, and run the remaining data from there.

Suppose that our clone detector has a bug in `RoughCloner`, the second stage of the algorithm. Once we've noticed the bug, we can re-execute the first stage of the algorithm, but this time, write the result into a temporary table:

```
HadoopPipeline<Repo, Snippet> pipe = HadoopPipeline.fromTo(inTable, tmpTable);
pipe
  .influx(new RepoCodec())
  .mapAndEfflux(new SnippetExtractor(), new SnippetCodec());
```

Now, to re-run `RoughCloner`, all we have to do is start a pipeline from there. Note that nothing forces us to make that pipeline a Hadoop pipeline. This is how we can start a local pipeline that reads its data from a Bigtable across the network. Local pipelines can be debugged like any local-running programs, right from within the IDE.

```
LocalPipeline<Snippet, Clone> localPipe = LocalPipeline.from(tmpTable.asCellSource());
pipe
  .influx(new SnippetCodec())
  .mapAndEfflux(new RoughCloner(), new CloneCodec());
```

Here, the method `asCellSource()` returns a cell source that reads directly from a Bigtable, through the network. The resulting pipeline will pull the data out of the Bigtable, to our local work station, and execute the job in memory, shuffling in memory. If the data size is too big to be dealt with in memory, we can, for debugging, add a filter around the cell source that will only read the first few thousand cells.

6.2.2 Independent of MapReduce

We want to allow developers to postpone the decision of whether to run locally, or in a distributed fashion, as long as possible. Therefore, Cells does not depend on a running MapReduce cluster. It does not even depend on any MapReduce packages. The cells library is split into two packages, `cells` and `cells.hadoop`. While `cells.hadoop` depends on Hadoop classes, `cells` is a from-scratch implementation of the MapReduce paradigm, including its own implementation of in-memory Bigtables.

We are careful about the distinction because Cells is a general-purpose library for writing parallel programs, even if no distributed execution is ever intended. Cells, without the Hadoop package, and without the Hadoop libraries, consists of a total of 2105 lines, including code comments. In contrast, if cells were to depend on all of Hadoop and HBase, any program using Cells would grow in size by hundreds of megabytes.

6.2.3 Static type checking

Keeping track of data encoding while reading from and writing to HBase tables is fragile and dangerous. Since HBase stores only raw bytes, careful programming is needed to make encoding and decoding match up, as they're often part of different mappers and reducers. In an early version of the clone detector that was made up of hand-written Hadoop jobs, a significant share of the code was used merely for checking if the encoding of one stage matched the decoding in another stage.

Cells addresses this by using generics. In the following snippet, we added casts to show the generic types. Here, `Mapper<Repo, Snippet>` means a mapper that accepts rows of `Repo`s, and outputs cells of `Snippet`s.

```
void run(Pipeline<Repo, Clone> pipe) {
  pipe
    .influx((Codec<Repo>) new RepoCodec())
    .map((Mapper<Repo, Snippet>) new SnippetExtractor())
    .shuffle((Codec<Snippet>) new SnippetCodec())
    .mapAndEfflux((Mapper<Snippet, Clone>) new RoughCloner(),
      (Codec<Clone>) new CloneCodec());
}
```

All of these types are checked by Java's type system. For example, `mapAndEfflux` must be called with a codec of generic type `Clone`, because the type of the pipeline says that the output is of type `Clone`. Even more, arbitrarily long chains can be statically type checked, so that the output type of one mapper must match the input type of the following mapper, and the codec in between.

6.2.4 Predictable performance

The close correspondence between the computational model of Cells and the implementation makes performance predictable and controllable. Since shuffling is a comparatively expensive operation, the developer is well advised to shuffle as few times as possible. The flip side of this is the lack of automatic optimizations. We see the usefulness of query optimizers, but we don't think that Cells is the right level of abstraction to feature one.

6.3 Implementation

After designing the API of Cells to closely match the computational model of MapReduce and Bigtable, the implementation of a `HadoopPipeline` and an `Table` implementation that stores in `HBase` is straightforward. Therefore, in this section we will explain the implementation of the local and in memory part of Cells.

6.3.1 Sharding and map execution

To run mappers in parallel, the input must be broken into separately executable units, called *shards*. While in our computational model the smallest executable unit is a row, it can pay off to group more than one row into a shard. That is because scheduling shards is single-threaded, and we run a risk of under-using our parallel capabilities. Cells vaguely defines a good shard size as big enough to avoid spending much time scheduling, small enough to avoid individual shards taking much more time than others.

Since the definition of a good sharding depends on the size of the input cells as much as on their number, sharding is best dealt with early in the pipeline. The user can control sharding by implementing his own `CellSource` – the input of a `LocalPipeline`. For example, a `CellSource` that reads files from a disk may choose to group files into a shard, until each shard contains at least one Megabyte of data, even if the `CellSource` converts each file into its own cell, with one cell per row.

To execute a pipeline stage, we clone its mapper until we have as many mappers as we have threads in our thread pool. Furthermore, we create empty output shards, as many as we have input shards, to receive the output of running mappers. In a loop, we pick up a mapper and unprocessed shard, and schedule the execution in the thread pool. Mappers can be recycled for computing more than one shard. Because mappers write to an uncontested output shard, mapper execution is completely lock-free.

6.3.2 Lock-free shuffle

After the execution of mappers, as seen in section §6.3.1, the mapper output is in different shards, without any ordering or grouping. Therefore, before the next mapper can run, we have to bring all output shards back into order: All cells inside a shard must be ordered, and for any two shards, all cells in one of them must be smaller than all cells in the other. Furthermore, after ordering, a row (several cells of identical row key) must never be split between shards.

We achieve this using a variant of Sample Sort [30], similar to the variant by Helman *et al.* [36]. The algorithm works as follows for input *inShards*, a list of shards, and output *outShards*, where `SAMPLE_SIZE` is some integer constant.

1. From each shard in *inShards*, we randomly extract `SAMPLE_SIZE` cells into a sample. Single-threaded, we sort the sample, and take every kth element into a list named *splitters*, so that there is one more element in *splitters* than there are shards. Furthermore, we set the column key of all cells in *splitters* to be empty to ensures that rows cannot be split between shards. The first element in splitters must be smaller or equal than all cells, the last should greater than all cells.

2. In parallel, copy cells from*inShards* into a new list of shards, *outShards*, so that all cells in *outShards*[i] are greater or equal than *splitters*[i], but smaller than *splitters*[i+1].

3. In parallel, sort all *outShards*.

Here, only the sampling phase is single-threaded. The other phases are fully parallel and lock-free. The time spent in sampling depends wholly on the size of the sample, controlled by the constant `SAMPLE_SIZE`. We chose the value empirically, by running `WordCount` with a range of different values, as seen in figure 6.1. The best value size turned out to be 16. There are larger values that yield the same performance, but on very small pipelines a smaller value is obviously better.

6.3.3 In-memory Bigtable

For side inputs and outputs, Cells offers tables, which are directly modeled after Bigtable [14]. Since the subtleties of the Bigtable model all involve data distribution and replication, an in-memory implementation is straightforward. To further simplify matters, tables cannot be both written and read during the same pipeline stage. This means that we can write, sort and shuffle them exactly like the main output at the end of a pipeline stage, as described in section §6.3.2.

To read the table again, the data is offered through the `LookupTable` interface, outlined in section §6.1.2. The implementation of lookups in `LookupTable` is simple enough: since cells are sorted inside each shard, and since shards are sorted too, to lookup a row key, we can first binary search for its shard, and then binary search inside of the shard, to find its row. To lookup by column, we maintain an index, exactly as we do in section §6.3.4.

6.3.4 Column-lookup for HBase

HBase doesn't support lookup by column key [32], unlike Bigtable [14]. To support lookup by column key, Cells maintains an index over every HBase table.

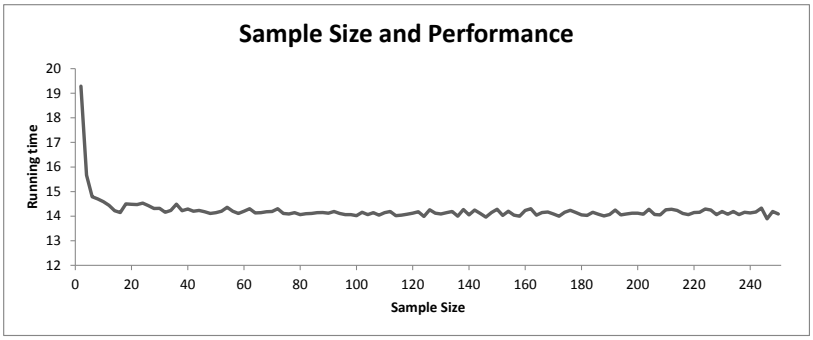


Figure 6.1: Run time, in seconds, as function of `SAMPLE_SIZE`.

In HBase, columns can be kept in different namespaces, using 'column families'. Whenever a cell is written into an `HBaseSink`, we store an additional cell in the 'index' column family, which has the row and column keys flipped, but no cell contents.

Using this structure, we can efficiently look up column keys. We simply look up the column key in the index, retrieving all row keys that contain cells with this column key. Then, we can lookup the actual cells by looking up the retrieved row keys with the column key.

6.4 Benchmarks

We chose two problems to test the performance of Cells. The first is `WordCount`, which makes for a great benchmark because the operation inside of the Mappers is relatively inexpensive, leading us to mostly measure the time spent in library code. The second is distributed Support Vector Machine training, which is a more realistic benchmark, as it is much more computationally demanding, but can easily be expressed as a MapReduce job [92] . We compare Cells' performance both with plain Hadoop and PLINQ. All benchmark code can be found on Github, together with the project sources[4].

For `WordCount`, we sampled a number of books from project Gutenberg[5] — 600 files, 240mb in total. The data is available together with the project sources. For SVM training, we use a dataset[6] provided to us by Yahoo with around 40

[4]Link removed for anonymity
[5]`http://www.gutenberg.org`
[6]`http://webscope.sandbox.yahoo.com/catalog.php?datatype=r`

million log lines of user interactions with Yahoo's front page.

Our local benchmarks were run on an 2.4 GHz 8 core i7 PC with 16Gb of memory. Our cluster setup consists of three server machines, with 8, 8 and 32 cores, and 16, 16 and 120 GB of RAM, respectively. The code of benchmarks is available on GitHub, together with the Cells sources. We make a sequence of 50 runs and measure the minimum time taken. Using the minimum helps us mitigate VM-related blunders.

Table 6.1: Cells vs PLINQ Benchmark in seconds.

	Cells	PLINQ
WordCount	14.3	26.5
SVM	261.1	N/A

On `WordCount`, Cells is almost twice as fast as PLINQ. This is somewhat surprising because our approach by design, during shuffling, brings the cells into a well-specified ordering that is irrelevant to counting words. The benchmark results indicate that this design choice does not hurt performance.

We couldn't run the SVM training problem on PLINQ due to the excessive memory consumption, leading, eventually to OutOfMemoryException.

Table 6.2: Cells vs Hadoop Benchmark in seconds.

	Cells	plain Hadoop
WordCount	411.7	392.2
SVM	437.02	413.13

On both problems, Cells is on par with the plain Hadoop version for distributed execution. Cells uses Hadoop to run distributed MapReduce job, so it would be surprising to outperform hand-written Hadoop. However, the price for using Cells is small.

The difference in efficiency between in-memory execution and execution in Hadoop is dramatic, but not unusual. Hadoop's poor efficiency is well-known [1, 65].

6.5 Related work

Lee *et al.* [57] survey how MapReduce is used for parallel data processing.

Compared to MapReduce [25], Cells offers side inputs, side outputs, the ability to run jobs locally, and an abstraction for inputs and outputs that is independent of their storage, simplifying testing and debugging drastically. Stock MapReduce offers no support for pipelines, and running a Reduce must always be preceded by running a Map stage.

FlumeJava [13] is a framework using a flavor of functional programming that is automatically executed either locally, or on MapReduce. FlumeJava is a complex framework, automatically combining as many operations as possible into the map and the reduce stages, and automatically estimating whether jobs should be executed locally or remotely. Unlike Cells, FlumeJava does not require

its computation to be a pipeline, but instead allows arbitrary DAGs. In contrast, Cells is a simple library, only about 4500 lines, including comments, with completely predictable performance, in a style that is not specific to parallel execution.

PLINQ [38] is an extension of LINQ [61] that mixes imperative with functional programming to automatically determine which parts of a program can be run in parallel. PLINQ programs aren't restricted to MapReduce style. Since the level of abstraction of expressing computations is high, PLINQ makes it hard to predict which, if any, parts of a program will run in parallel. If PLINQ fails to detect a parallelizable execution, it will simply run single-threaded. PLINQ supports arbitrary data flow graphs, which do not have to be known to the framework upfront. In practice, this means that PLINQ executes everything as lazily as possible, and avoids throwing away intermediate results, in case they will be needed again. In contrast, Cells' simple computational model makes the implementation very easy, and allows us to throw away intermediate results of a shuffle once the following mapper has read them completely. This leads to Cells using significantly less RAM than PLINQ. In our benchmarking, a job that required only 4 GB of RAM in Cells could not be made to run using 7 GB of RAM in PLINQ.

Dryad [41] allows more more complex computational structures than just pipelines. Directed acyclic graphs are supported, and there is a great deal of liberty in how the individual stages are executed: even SQL queries can be executed as part of a computational graph. Configuration is therefore necessarily more complex, up to the point where Dryad features a separate tool just to help configure computational graphs. By contrast, Cells is simple and small, with a computational model that makes performance predictable.

Various high-level tools build on MapReduce to distribute execution. Examples include Apache Pig [64] and Apache Hive [88]. All translate declarative queries automatically into MapReduce jobs. All of these tools directly generate distributed jobs. If, instead, they were to generate Cells jobs, this would simplify testing of the generated jobs, without incurring significant cost in performance on the distributed execution.

6.6 Conclusion

Cells is a simple and small library for writing parallel jobs. For local execution, it can even outperform PLINQ by factor 2 on the WordCount benchmark. For distributed execution, it adds only a small overhead to the execution. It makes parallel and distributed computation easier to test and write.

We wrote Cells after having written the entire clone detector as raw MapReduce jobs already. The problems that Cells solves: better debuggability, cleaner support for coding and decoding, were very much exactly the problems we had while implementing our clone detector without it.

Acknowledgements

This chapter is based on work I did with Alexey Kolesnichenko and Oscar Nierstrasz [76].

Chapter 7

Clone detector

In chapter 4, we found clones in a big repository, but our suggested approach was not yet *scaleable*, *i.e.,* could handle growing amounts of data without changing the approach [9]. That is because we have failed to show a scheme to distribute our computation such that it can handle growing load by enlarging the system.

We improve on our detector in chapter 4 in the following ways. Rather than produce only one hash per function, we now produce one per line. This greatly improves recall, as it now becomes possible to detect clones that are only a fraction of a function. Furthermore, we will add language-specific parsing to our approach. Rieger [71] shows that minimal parsing improves recall of clone detectors. Finally, we describe how to employ MapReduce pipelines to compute the clones fully distributed in mappers that are data-local, *i.e.,* without lookups across the cluster, and quasilinear, and therefore fully scaleable.

Our clone detector achieves scaleability by combining the cornerstones we've seen before: we use bad hashing to avoid a clustering phase and build an algorithm where all stages are data-local, regular expressions to achieve quasilinear run time, and MapReduce pipelines to distribute the computation.

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 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.

In this chapter, we describe a clone detector that is conceptually similar to conQAT [40] (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 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 below 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 known Java frameworks and libraries, in all tagged versions. We detect code clones across 16,953,815 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.

7.1 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 [44]. 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.

These rules require some parsing of the sources, which we can achieve via lightweight parsing using regular expressions, as we have discussed in chapter 5. This ensures that the computation of a bad hash is linear in time. We can achieve all replacing in just one pass, by extracting all the relevant information in just one pass, by setting the grammar complex enough. Then, walking the parse tree, we output the transformed tree, similar to how Structural Regular Expressions[68] work.

Let a 'snippet' be a run of five consecutive lines of code. We use the same bad hashing as in section §4.1, but now applied to snippets rather than entire methods. 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) were 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

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

sequences of digits are replaced with the number "1". For an example, see Table 4.1.

- Two documents are detected as type-3 clones if and only if they share the same sketch.

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 [78], as seen in chapter 4. However, the precision of type-1 clones is very high, given that they are exact clones, which rarely occur by chance. Since there are, as a percentage, only few more type-3 clones than type-1 clones, precision of type-3 clones 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 7.1. 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 7.2.

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 7.1, 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 7.1, 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 smaller than 85 % of the size of their union, we discard the clone. The latter rule has been proposed and validated by Koschke [52]. 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;
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();
                  }
                }
```

Figure 7.1: 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 3 snippets, FUN2 and FUN3 collide in 11 snippets. The frames show the merged snippet collisions, which we call *clones*.

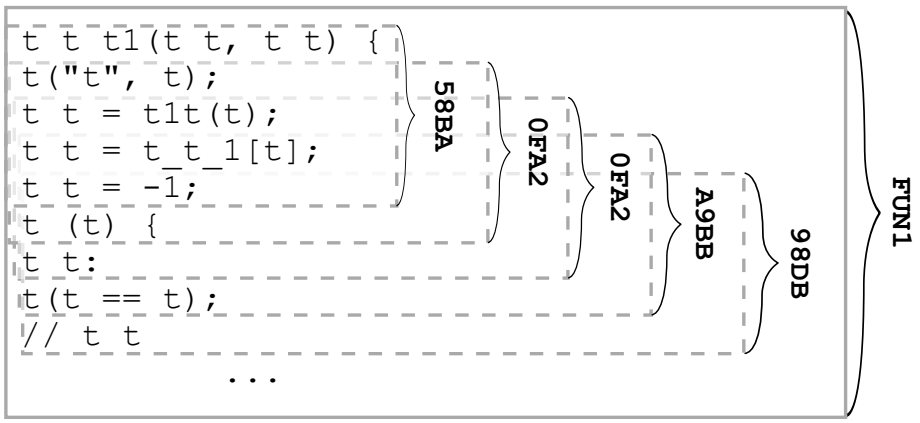


Figure 7.2: The first few snippets and their hashes, for the first function in Figure 7.1.

7.2 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 [86] and Unix pipes. The second part is implemented in Java and achieves parallelism using a MapReduce pipeline.

7.2.1 Mining the Internet for source code

In this section, each paragraph 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 [86]. 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.

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.

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.

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.

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.

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

Figure 7.3: 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.

Move local repositories into distributed file system. 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 7.3 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.

7.2.2 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. 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 7.4. Let us discuss all mappers in turn.

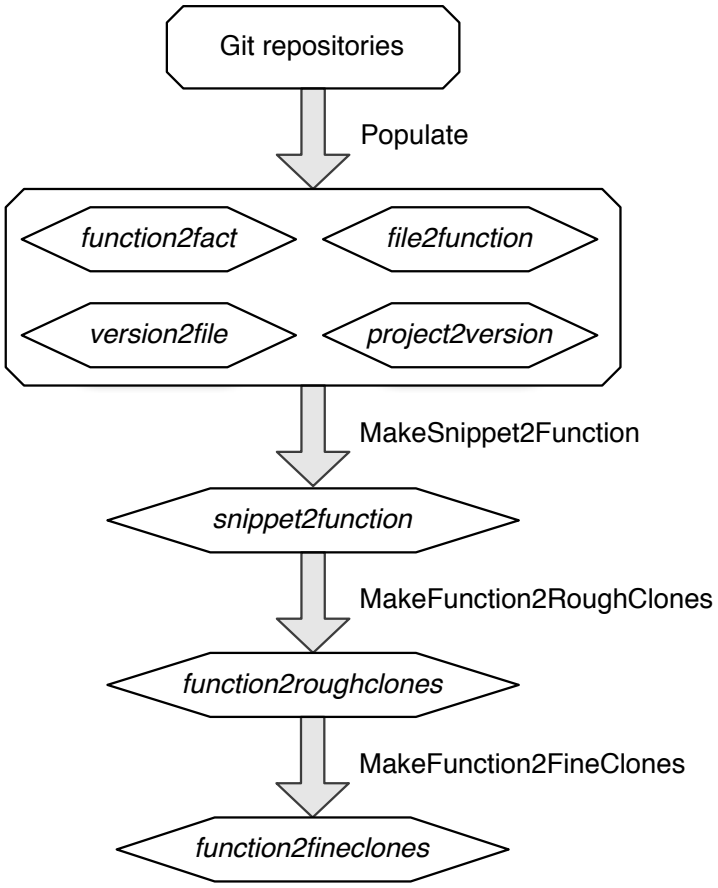


Figure 7.4: The pipeline architecture of our algorithm.

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 [27], 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 7.6.

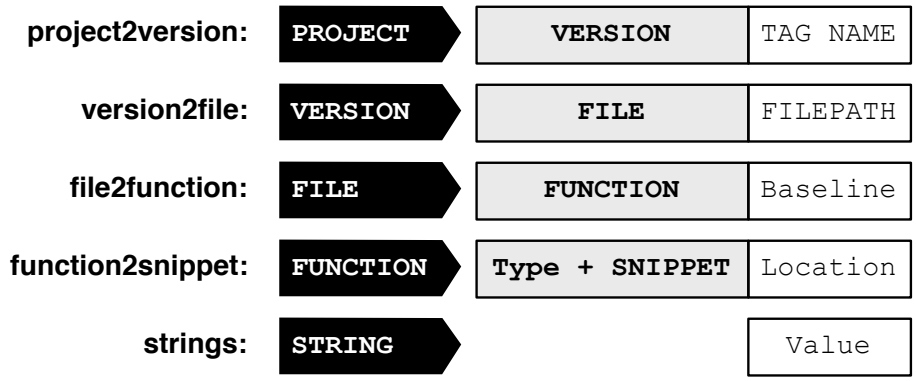


Figure 7.5: The table design for storing all repositories using the flyweight pattern. Names printed in all caps are hashes.

This mapper produces several tables whose layout can be seen in Figure 7.5. 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 are 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 are stored exactly once, in the 'strings' table. This is illustrated in figure 7.6.

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 from chapter 5, 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.*?\{`.

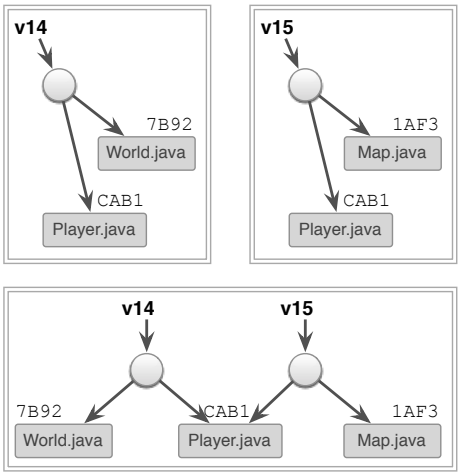


Figure 7.6: Versions v14 and v15 (possibly from different projects!) both contain a file named Player.java, whose contents has hash CAB1 (above). We save space by forcing v14 and v15 to share Player.java, thus preventing the duplication of Player.java in our database.

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 regular expression 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 7.1, the main output is seen in Table 7.1.

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 7.1. 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

Table 7.1: 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		

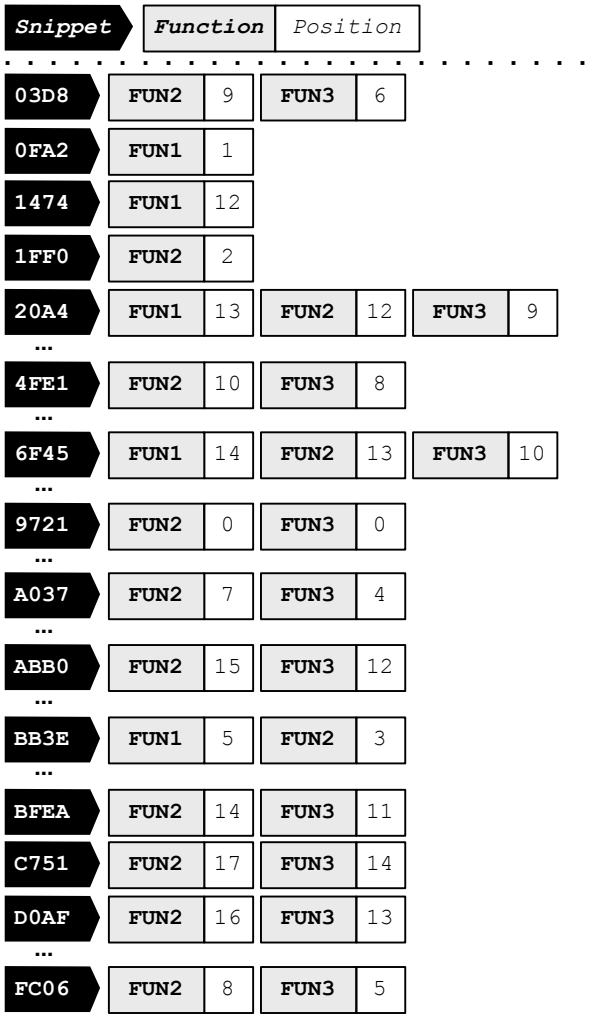
size, snappy-compressed.

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.

Rough clones In this mapper, we gather, for every function, all collisions with it. For example, in the example in Table 7.3, 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 pseudocode of this operation in Figure 7.7, the output of this mapper is of size $O(c^2 n)$, where

Table 7.2: Output of the mapper from Paragraph §7.2.2. Row keys are snippets, cell values are functions that contain these snippets.



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 7.8, we have to introduce special treatment to snippets that occur in very many functions, thus providing an upper bound for c. As seen in Figure 7.8, 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 1,000 functions are still written out, 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[2] also matches. This serves as a useful filter: if a snippet is extremely common, it prob-

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

```
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)
    }
  }
}
```

Figure 7.7: The pseudocode that produces Table 7.3 from input Table 7.2

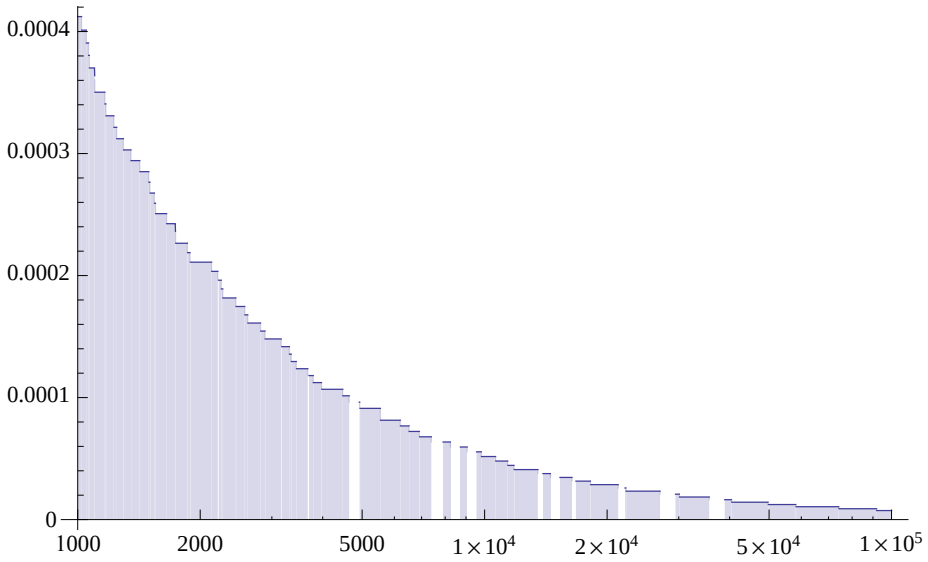


Figure 7.8: Percentage of snippets that have more than n clones. The y-axis shows the percentage, the x-axis shows the number of clones. If a snippet occurs in two places, it has two clones. The x-axis is shown in range $[1000, 100000]$.

ably 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 is seen in Tables 7.3 and 7.2.

Table 7.3: 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*		*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		

Table 7.4: The content of the popular snippets Table. For our data set, this table is 293.3 MB in size.

Function	*Snippet*	*Position*	*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 %.

Clone expansion The input to this mapper is seen in Table 7.3: 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 7.4 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 7.5. Going back to the input files in figure 7.1, we see that we have detected, meaningful and intuitive clones for this input.

Table 7.5: 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.

7.3 Discussion

The time to detect clones across versions and across all Java-projects on Ohloh can be found in Table 7.6. The key ingredients to its performance are twofold. 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.

7.3.1 Precision and Recall

We have yet to measure precision and recall of our algorithm, but since our normalizations are the same as those used by CCFinder [44], our approach should provide roughly the same numbers (CCFinder achieves 40 % precision and 40 % recall in Bellon *et al.* 's study [8]), or perhaps better, since we added Koschke's filtering [52] of false positives to the approach.

Table 7.6: 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

7.3.2 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[3], 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.

7.3.3 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.

At first, for only 405 repositories, stage "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.

7.3.4 Future work

We have outlined in chapter 1 that clone detection can serve as a platform to build tools to connect projects. We are in a great position to build these tools.

We have already implemented one to group clones into groups, to be able to better display our results. It works as follows. We define functions to be the vertices of a graph, and clones the edges on it. Then, for each connected component, the set of all vertices is defined as one clone group. We have implemented this in a non-parallel way, simply because the data volume of filtered, interesting clones is small enough to permit it, even for our input size.

We have not yet measured precision and recall of our approach. Precision could be measured by deciding letting a group of experts mark, for each clone, whether or not it is interesting. Recall could be measured by running our clone detector on Bellon's data [7], which will require us to implement a C++ frontend to our detector.

7.4 Conclusion

Our clone detector combines bad hashing, lightweight 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.

Our clone detector achieves scale through quasilinear run-time, distribution, and data locality. We achieve them by relying three cornerstones. Lightweight parsing using regular expressions lets us parse all input in one pass in linear time.

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

MapReduce pipelines let us distribute a computation to an arbitrary number of machines, and therefore makes our algorithm scaleable, so long as all computation is data local. Bad Hashing lets our clone detector not require a clustering phase, and therefore make it easy to keep all computation data local.

Acknowledgements

This chapter is based on work I did with Simon Vogt and Oscar Nierstrasz. Simon wrote substantial parts of the pipeline, and the parts he did not write, he code reviewed.

Bibliography

[1] Eric Anderson and Joseph Tucek. Efficiency matters! *SIGOPS Oper. Syst. Rev.*, 44(1):40–45, March 2010. doi:10.1145/1740390.1740400.

[2] Arvind Arasu, Surajit Chaudhuri, Zhimin Chen, Kris Ganjam, Raghav Kaushik, and Vivek R. Narasayya. Experiences with using data cleaning technology for bing services. *IEEE Data Eng. Bull.*, 35(2):14–23, 2012.

[3] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second IEEE Working Conference on Reverse Engineering (WCRE)*, pages 86–95, July 1995.

[4] 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. doi:10.1109/ICSM.1998.738528.

[5] Andrew Begel and Robert DeLine. Codebook: Social networking over code. In *ICSE Companion*, pages 263–266, 2009. doi:10.1109/ICSE-COMPANION.2009.5070997.

[6] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM. doi:10.1145/1806799.1806821.

[7] Stefan Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis, Universität Stuttgart, September 2002. URL: http://www.bauhaus-stuttgart.de/bauhaus/papers/DIP-1998.pdfhttp://www.bauhaus-stuttgart.de/clones/index.html.

[8] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007. doi:10.1109/TSE.2007.70725.

[9] André B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 195–203, New York, NY, USA, 2000. ACM. doi:10.1145/350391.350432.

[10] Romain Brixtel, Mathieu Fontaine, Boris Lesner, Cyril Bazin, and Romain Robbes. Language-Independent clone detection applied to plagiarism detection. In *SCAM*, pages 77–86, 2010. doi:10.1109/SCAM.2010.19.

[11] 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. IEEE Computer Society, June 1997.

[12] Paul Cairns and Anna L. Cox. *Research Methods for Human-Computer Interactions*. Cambridge University Press, 2008. Chapter 2, 7, 9.

[13] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. *SIGPLAN Not.*, 45(6):363–375, June 2010. doi:10.1145/1809028.1806638.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, June 2008. doi:10.1145/1365815.1365816.

[15] 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. doi:10.1145/1370750.1370766.

[16] Jonathan Cloud and Graham M. Vaughan. Using balanced scales to control acquiescence. *Sociometry*, 33(2):pp. 193–202, 1970. URL: http://www.jstor.org/stable/2786329.

[17] James R. Cordy. Comprehending reality—practical barriers to industrial adoption of software maintenance automation. In *Proc. 11th Int. Workshop on Program Comprehension (IWPC'03)*, pages 196–205, Portland, Oregon, USA, May 2003. IEEE.

[18] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel, 248966-028 edition, July 2013.

[19] R. Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...). *URL: http://swtch.com/~rsc/regexp/regexp1.html*, 2007.

[20] R. Cox. Regular expression matching: the virtual machine approach. *URL: http://swtch.com/~rsc/regexp/regexp2.html*, 2009.

[21] R. Cox. Regular expression matching in the wild. *URL: http://swtch.com/~rsc/regexp/regexp3.html*, 2010.

[22] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM. doi:10.1145/2145204.2145396.

[23] Michel Dagenais, Ettore Merlo, Bruno Laguë, and Daniel Proulx. Clones occurrence in large object oriented software packages. In *Proceedings of CASCON 1998*, pages 192–200, 1998.

[24] 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. doi:doi.acm.org/10.1145/1985441.1985468.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. doi:10.1145/1327452.1327492.

[26] 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.

[27] 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.

[28] Danny Dubé and Marc Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 2000. doi:10.1007/s002360000037.

[29] Ekwa D. Ekoko and Martin P. Robillard. Clonetracker: tool support for code clone management. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 843–846, New York, NY, USA, 2008. ACM. URL: http://dx.doi.org/10.1145/1368088.1368218, doi:10.1145/1368088.1368218.

[30] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, July 1970. doi:10.1145/321592.321600.

[31] Richard A. Frost and Barbara Szydlowski. Memoizing purely functional top-down backtracking language processors. *Science of Computer Programming*, 27(3):263–288, November 1996. doi:10.1016/0167-6423(96)00014-7.

[32] Lars George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[33] Niels Bjørn Bugge Grathwohl, Fritz Henglein, Lasse Nielsen, and Ulrik Terp Rasmussen. Two-Pass greedy regular expression parsing. In Stavros Konstantinidis, editor, *Implementation and Application of Automata*, volume 7982 of *Lecture Notes in Computer Science*, pages 60–71. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-39274-0_7.

[34] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proceedings of the 1st Workshop on Ecosystem Architectures*, pages 1–5, 2013. URL: http://scg.unibe.ch/archive/papers/Haen13a-EcosystemInformationNeeds.pdf.

[35] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24:8–12, 2009. doi:10.1109/MIS.2009.36.

[36] David R. Helman, David A. Bader, and Joseph Jájá. A randomized parallel sorting algorithm with an experimental study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, July 1998. doi:10.1006/jpdc.1998.1462.

[37] Rich Hickey. The Clojure programming language. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–1, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1408681.1408682, doi:10.1145/1408681.1408682.

[38] Gastn Hillar. *Professional Parallel Programming with C#: Master Parallel Extensions with .NET 4*. Wrox Press Ltd., Birmingham, UK, UK, 2010.

[39] Daqing Hou, Patricia Jablonski, and Ferosh Jacob. Cnp: Towards an environment for the proactive management of copy-and-paste programming. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 238–242. IEEE, May 2009. URL: http://dx.doi.org/10.1109/ICPC.2009.5090049, doi:10.1109/ICPC.2009.5090049.

[40] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *ICSM*, pages 1–9, 2010. doi:10.1109/ICSM.2010.5609665.

[41] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM. doi:10.1145/1272996.1273005.

[42] Adam Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009. doi:10.1145/1536616.1536632.

[43] Stan Jarzabek and Li Shubiao. Eliminating redundancies with a 'composition with adaptation' meta-programming technique. In *Proceedings ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 237–246. ACM Press, September 2003.

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

[45] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. *WCRE '06*, 0:19–28, 2006. URL: http://dx.doi.org/10.1109/WCRE.2006.1, doi:10.1109/WCRE.2006.1.

[46] L. Karttunen, J. P. Chanod, G. Grefenstette, A. Schiller, and Received February. Regular expressions for language engineering. In *Natural Language Engineering*, pages 305–328, 1996. doi:10.1.1.28.4880.

[47] Steven M. Kearns. Extending regular expressions with context operators and parse extraction. *Softw: Pract. Exper.*, 21(8):787–804, August 1991. doi:10.1002/spe.4380210803.

[48] Iman Keivanloo, Juergen Rilling, and Philippe Charland. Internet-scale real-time code clone search via multi-level indexing. In *WCRE*, pages 23–27, 2011. doi:10.1109/WCRE.2011.13.

[49] Miryung Kim, Vibha Sazawal, David Notkin, and Gail C. Murphy. An empirical study of code clone genealogies. In *Proceedings of European Software Engineering Conference (ESEC/FSE 2005)*, pages 187–196, New York NY, 2005. ACM Press. doi:10.1145/1081706.1081737.

[50] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society. doi:10.1109/ICSE.2007.45.

[51] Rainer Koschke. Identifying and removing software clones. In *Software Evolution*, chapter 2, pages 15–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. URL: http://dx.doi.org/10.1007/978-3-540-76440-3_2, doi:10.1007/978-3-540-76440-3_2.

[52] 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. doi:10.1109/csmr.2012.37.

[53] Jens Krinke, Nicolas Gold, Yue Jia, and David Binkley. Cloning and copying between gnome projects. In *MSR*, pages 98–101, 2010. doi:10.1109/MSR.2010.5463290.

[54] Chris Kuklewicz. Regular expressions/bounded space proposal, February 2007.

[55] V. Laurikari. Nfas with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 181–187. IEEE, 2000.

[56] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research methods in Human-Computer Interaction*. Wiley, 2010.

[57] Kyong H. Lee, Yoon J. Lee, Hyunsik Choi, Yon D. Chung, and Bongki Moon. Parallel data processing with MapReduce: A survey. *SIGMOD Rec.*, 40(4):11–20, January 2012. doi:10.1145/2094114.2094118.

[58] Rensis Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[59] 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.

[60] Xin Lu. *Respondent-Driven Sampling: Theory, Limitations & Improvements*. PhD thesis, Karolinska Institutet, 2013.

[61] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM. doi:10.1145/1142473.1142552.

[62] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, page 8, Berkeley, CA, USA, 2010. USENIX Association. URL: http://portal.acm.org/citation.cfm?id=1929831.

[63] Lasse Nielsen and Fritz Henglein. Bit-coded regular expression parsing. In Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, volume 6638 of *Lecture Notes in Computer Science*, pages 402–413. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-21254-3_32.

[64] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM. doi:10.1145/1376616.1376726.

[65] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM. doi:10.1145/1559845.1559865.

[66] Shaun Phillips, Guenther Ruhe, and Jonathan Sillito. Information needs for integration decisions in the release process of large-scale parallel development. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1371–1380. ACM, 2012.

[67] R. Pike. The text editor sam. *Software: Practice and Experience*, 17(11):813–845, 1987.

[68] Rob Pike. Structural regular expressions. In *Proc. EUUG Spring Conf., Helsinki 1987*, 1987.

[69] POSIX.1-2008. The open group base specifications. Also published as IEEE Std 1003.1-2008, July 2008. doi:10.1109/IEEESTD.2008.4694976.

[70] S.P. Reiss. Visualizing the java heap demonstration proposal. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 389 –390, September 2009. doi:10.1109/ICSM.2009.5306287.

[71] Matthias Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, University of Bern, June 2005. URL: http://scg.unibe.ch/archive/phd/rieger-phd.pdf.

[72] Mary Beth Rosson and John M. Carroll. Active programming strategies in reuse. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 4–20, Kaiserslautern, Germany, July 1993. Springer-Verlag. URL: http://link.springer.de/link/service/series/0558/tocs/t0707.htm.

[73] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74:470–495, May 2009.

[74] Satu E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, August 2007. doi:10.1016/j.cosrev.2007.05.001.

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

[76] 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.

[77] Niko Schwarz, Mircea Lungu, and Oscar Nierstrasz. Seuss: Decoupling responsibilities from static methods for fine-grained configurability. *Journal of Object Technology*, 11(1), 2012. URL: http://www.jot.fm/issues/issue_2012_04/article3.pdf, doi:10.5381/jot.2012.11.1.a3.

[78] 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. URL: http://scg.unibe.ch/archive/papers/Schw12a-scalable-clones.pdf.

[79] Niko Schwarz, Erwann Wernli, and Adrian Kuhn. Hot clones, maintaining a link between software clones across repositories. In *IWSC '10: Proceedings of the 4th International Workshop on Software Clones*, pages 81–82, New York, NY, USA, April 2010. ACM. URL: http://scg.unibe.ch/archive/papers/Schw10b-hot-clones.pdf, doi:10.1145/1808901.1808915.

[80] Robert Sedgewick. *Algorithms in C (paperback)*. Addison-Wesley Professional, 1 edition, January 1990. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0768682339.

[81] Dominik Seichter, Deepak Dhungana, Andreas Pleuss, and Benedikt Hauptmann. Knowledge management in software ecosystems: software artefacts as first-class citizens. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 119–126. ACM, 2010.

[82] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM. URL: http://people.cs.ubc.ca/~murphy/papers/other/asking-answering-fse06.pdf, doi:10.1145/1181775.1181779.

[83] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2 edition, February 2005. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0534950973.

[84] Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications Inc., 1998.

[85] M. Sulzmann and K.Z.M. Lu. Regular expression sub-matching using partial derivatives. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 79–90. ACM, 2012.

[86] O. Tange. GNU parallel - the Command-Line power tool. *;login: The USENIX Magazine*, 36(1):42–47, February 2011. URL: http://www.gnu.org/s/parallel.

[87] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. doi:10.1145/363347.363387.

[88] Ashish Thusoo, Joydeep S. Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive- a warehousing solution over a Map-Reduce framework. In *IN VLDB '09: PROCEEDINGS OF THE VLDB ENDOWMENT*, pages 1626–1629, 2009. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.151.2637.

[89] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dx.doi.org/10.1109/VLHCC.2004.35, doi:10.1109/VLHCC.2004.35.

[90] 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. doi:doi.ieeecomputersociety.org/10.1109/WCRE.2011.12.

[91] Filip van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proc. 19. Intl. Conference on Automated Software Engineering (ASE'04)*. IEEE, September 2004.

[92] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname: .....

Matrikelnummer: .....

Studiengang: ……

Bachelor ☐ Master ☐ Dissertation ☐

Titel der Arbeit: .....

.....

.....

LeiterIn der Arbeit: .....

.....

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

.....

Ort/Datum

.....

Unterschrift

Curriculum Vitae

Personal Information

Name:	Niko Schwarz
Date of Birth:	8 September, 1983
Place of Birth:	Heidelberg, Germany
Nationality:	Germany

Education

2009–2014 **PhD in Computer Science**
Software Composition Group,
University of Bern, Switzerland
`http://scg.unibe.ch`

2005–2009 **M. Sc. in Mathematics**
University of Jena, Germany.
Thesis title: Rank aggregation by criteria