Analyzing Large Code Repositories

by

Petr Maj

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Dissertation degree study programme: Informatics
Department of Theoretical Computer Science

Prague, December 2022

Supervisor:
Jan Janousek
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Co-Supervisor:
prof. Jan Vitek, Ph.D.
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Copyright © 2022 Petr Maj

Abstract and contributions

Mining software repositories is very important for large set of problems as thanks to open source, online collaboration platforms and version control systems, all of the information we really need is at our fingertips. The problem, as with any big data problem though is the signal to noise ratio as the data is extremely messy. This thesis analyzes the hurdles associated with large scale repository mining in the PL community context, proposes methodologies for correct processing of such data and the intended use and culminates in the design of a tool that aids researchers in the task.

In particular, the main contributions of the dissertation thesis are as follows:

1. Analysis of cloning and associated biases in large collections of software repositories (paper 1)
2. Analysis of reproducibility issues and statistical interpretation of large corpora. Proposal of better methodology for reproducibility (paper 2)
3. Design and implementation of a tool for large scale download, archival and querying of software repositories to aid reproducble project selection and analysis (pape 3)
4. Analysis of the selection bias introduced by the most frequentlty used project selection by popularity convenience sampling on recent papers. Analysis of the obtained and missed projects and development of methodology for reproducible and validable project selection and associated tooling (paper 4)

Keywords:
repository mining, big code, code duplication, selection bias.

As a collaborator of Petr Maj and a co-author of his papers, I agree with Petr Maj's authorship of the research results, as stated in this dissertation thesis.

. . . .
Konrad Siek Jan Vitek Jakub Zitny Alexander Kovalenko

Acknowledgements

First of all, I would like to express my gratitude to my dissertation thesis supervisors, Professor Jan Vitek and Docent Jan Janousek.

Special thanks go to the staff of the Department of Theoretical Computer ScienceDepartment of Theoretical Computer Science, who maintained a pleasant and flexible environment for my research.

My research has been supported by the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/15_003/0000421.

I would like to express thanks to my colleagues from the department, namely Mr. . . . , Ms. . . . , Dr. . . . , and others, for their valuable comments and proofreading.

Finally, my greatest thanks go to my family members, for their infinite patience and care.

Dedication

In memory of doc. Ing. Karel Müller, CSc. (1946? - 2011)

Contents

1 Introduction **1**
1.1 Motivation 2
1.2 Thesis 9
1.3 Structure of the Dissertation Thesis 10

2 Background and State-of-the-Art **11**
2.1 Sources 12
2.1.1 Github 12
2.1.2 Bitbucket 13
2.1.3 Other VCS Hosts 13
2.1.4 Package Managers 14
2.1.5 Software Heritage 14
2.2 Software Repositories 15
2.2.1 GitHub 16
2.2.2 Software Heritage 18
2.2.3 GH Torrent 19
2.2.4 Orion 20
2.2.5 Boa 20
2.2.6 Other Repositories 21
2.3 Summary 22

3 Overview of Our Approach **25**
3.1 Sources 25
3.2 GitHub Ecosystem 27
3.2.1 Duplication 27
3.2.2 Project Diversity 32
3.3 CodeDJ 32
3.3.1 Querying 34
3.3.2 Parasite 35

4 Main Results 39
4.1 Overview . . . 39
4.2 Paper 1 - DejaVu: A Map of Code Duplicates on GitHub . . . 41
4.2.1 Author Contributions . . . 41
4.3 Paper 2 - On the Impact of Programming Languages on Code Quality: A Reproduction Study . . . 42
4.3.1 Author Contributions . . . 42
4.4 Paper 3 - CodeDJ: Reproducible Queries over Large-Scale Software Repositories . . . 43
4.4.1 Author Contributions . . . 43
4.5 Paper 4 - The Fault in Our Stars: How to Design Reproducible Large-scale Code Analysis Experiments . . . 44
4.5.1 Author Contributions . . . 44
4.6 Discussion . . . 44
4.7 Summary . . . 44

5 Conclusions 45
5.1 Summary . . . 45
5.2 Contributions of the Dissertation Thesis . . . 45
5.3 Future Work . . . 45

Bibliography 47

Reviewed Publications of the Author Relevant to the Thesis 49

Remaining Publications of the Author Relevant to the Thesis 51

Remaining Publications of the Author 53

List of Figures

3.1 Data processing and analysis pipeline. 30
3.2 File versions broken into unique, original and copies. 31
3.3 Changes in GH projects over time 31
3.4 Lifespan of projects with at least one commit per year 32
3.5 Lifespan of projects with at least one commit per month 33

List of Tables

2.1 Comparison of major data sources and associated tooling. Repositories are classified in terms of their size, sources (primary menaning the repository maintains its own data store), whether they are still active, if and how a repository supports updates (not at all, regular dumps, or continuous), whether its queries are deterministic and reproducible 23

Chapter 1

Introduction

"The temptation to form premature theories upon insufficient data is the bane of our profession."

- Sherlock Holmes, fictional detective

The world as we know it relies upon billions of computers. From the tiny ones embedded in their hundreds in our cars, TV sets and even washing machines to the larger smartphones, laptops and computers, all the way to the very large supercomputers in datacenters, they are integral part of almost all aspects of the modern society. Over the years the size of computers has been reduced almosty as fast as their power increased: the smart watches some of us wear on their wrists are as powerful as desktop computers from the turn of the century which is about 100000 times faster than the computer that landed the Apollo missions on the moon.

True ingenuity of a computer comes from the fact that a single physical computer can perform different tasks depending only on its software, instructions that break the complex tasks the computer should perform to series of very simple operations the computer knows how to perform and describe the order in which those operations should be executed. To make a computer perform a new task then simply means to create a new software for it. But as computer systems increased their ubiquity and power, supplying all those computers with new software quickly and reliably become a problem, as Edsger Dijkstra famously said:

"The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

Edsger Dijkstra, The Humble Programmer (EWD340), Communications of the ACM

In other words, it is not the efficiency of the computer, or even the program itself, but the efficiency of the programmer in writing the programs that has become the bottleneck in software development process. The efficiency of a programmer, apart from their skill,

largely depends on two main factors: Programming languages and software development tools.

Programming languages As the software tasks shifted from simple number crunching of mathematical equations, the numeric values manipulated by computers become proxies of much more complex objects with more complicated relations. And while the low level instructions computers perform remain general enough to express those interactions, doing so resulted in laborious and error prone decomposion. The efficiency of a programmer suffered badly as instead of expressing the complex interactions that describe the intended functionality of a software, programmers spent most of their time in routine decomposition. To remedy this problem, high level programming languages were designed: Those languages abstract from the low level capabilities of the machine and focus on expressivity at levels better suited to the relations and algoritms of the more complex tasks. The decomposition previously done by a prgrammer is then delegated to automated tools such as compilers.

Software development tools As the size of software itself and the development teams responsible for its creation increased, actually typing the programs down become only a smart program of the software development process. Collaboration of multiple developers over extended periods of time become necessary and code reuse not just within a single project become more and more beneficial as the amounts of software available increased.

But despite the huge effort, the best intentions and bold claims of the authors, the goal of making programmers more efficient has been marred by almost complete lack of evaluation. There is very little facts we know about the effect of programming language design choices, or the tooling available on the productivity of programmers and quality of the programs. The absence of facts invites a sea of opinions so flame wars continously errupt over almost all aspects of software development, from benefits of programming language features, to superiority of programming styles, text editors, or whether to use spaces or tabs to visually structure the code.

1.1 Motivation

Let us concentrate on the first problem, which we can reduce into determining which of any two given programming languages is better - i.e. which language makes its programmers more effective. The hallmark of answering such questions scientifically is the controlled experiment: we run two experiments that are identical in their setup with the exception of a single variable. Any differences in their outcomes can thus be attributed to the single change with high certainty. To be more specific, we can ask two programmers to develop a web server in C++ and Haskell programming languages respectively. If the Haskell version will take less time to develop, we can concur that Haskell is the better language and vice versa.

But for a controlled experiment to work we must make sure that other variables that might influence the outcome are indeed controlled for. An obvious way would be to describe the programmer we are looking for as precisely as possible and then two programmers with the closest match from a larger pool of candidates. As we want to make sure that the

programmers have the same level of skill (both general programming and in their respective language), have same experience in the application domain the task will come from (i.e. we will not compare expert server programmer in C++ to an expert compiler programmer in Haskell) and have followed similar career choices (so that for instance we do not compare researchers to industry practitioners), the advertisement looking for the candidates might look like this:

> We are looking for a C++/Haskell programmers with exactly 5 years of profficiency in industry with a masters degree from computer science, or equivalent field and experience in server programming. Applicants are required to submit their full CV with special emphasis on their programming language and application development skills. The duration of the study is expected to be one month of full time work[1].

Although highly technical discipline, programming is heavily dependent on the human factor, which is next to impossible to control for. Even after carefully matching the resumes, the closest pair of programmers will differ in myriad of ways that cannot be infered from any of their submitted materials. Yet those differences, ranging from different upbringings and personal traits all the way to the quality of their computer science teachers and the first programming language they become comfortable with, may have substantial effect on their performance in the experiment. So when after a month-long programming session we would see one of the languages coming on top, instead of putting down the flame war between C++ and Haskell fans, we would have only poured more petrol in the fire.

Software engineering is not the only environment in which controlling for the human condition is necessary. Consider for instance the clinical trials that determine effectivity of a certain drug against a disease. Faced with the same impossibility to control for everything but the drug, the clinical trials establish two large groups of people: The control group that is not given any medication and the test group that is. The results for each of the groups are then averaged under the assumption that any variables that could not be controlled for will be similarly distributed in both the control and test groups and would therefore cancel each one out. In simple terms, if the test group recovers on average faster, the drug is effective even if there are a few people that actually recover slower, or even not at all. The larger samples effectively allow us to reclaim back the uncertainty introduced by the uncontrollable variables.

The same can be used for our programming languages problem. Instead of one programmer for each language, we can choose 100 programmers for each. We no longer have to pay attention to their better matching as long as we make sure the distributions of C++ and Haskell programmers with respect to the variables we can control for are similar, which is much easier task. If on average the Haskell programmers will be done sooner, then Haskell is indeed a better programming language, even if there will be a few real

[1]This is roughly how long it might take a single person to develop reasonably complex and feature-full web server for a meaningful comparison

programmers who would finish in C++ in a few days (only using the C subset of course). So after a month we would have the answer we wanted.

Except of course many C++ programmers would object saying that C++ has not been designed to be tool for writing web servers, for which other languages, such as Java are more suited to. To silence them we would rerun the experiment, but this time we would give different programming assignment. And if C++ wins this time, we will likely hear objections from the Haskell folks this time. Certainly there are many programs for which C++/Haskell were designed for and those might not even intersect. But let us not despair - we can use statistics again. Perhaps the actual programming task itself should also not be controlled - what if we give multiple different programming assignments and again average the results. Clearly then, the question will be settled once and for all.

Maybe. But now is the time to evaluate the practical (economical) feasibility of our endeavour: According to Statista[2] in 2019, Haskell and C/C++, although being the last two out of 10 languages with highest average salaries in the US averaged to $125000 and $121000 respectively, which rounded to $120k and converted to a monthly payment averages to a nice $10k per month. For our experiment, we require 200 man-months for a total cost of $2m for a single programming assignment, excluding the salaries of the involved researchers. Since this amount surpasses many reasearch grants themselves, it should be obvious that our proposal does not survive the reality and the win of C++ or Haskell remains as elusive as ever. And indeed for much of the relatively short history of the computer science, studies about the qualities of programming language design similar to our proposal were scattered far between and of very limited impact. Yet with more and more software being written and penetrating still larger parts of our lives, the problem of making programmers measurably more productive has only grown in importance.

The biggest hurdle is the practical impossibility to actually develop software in a controlled environment so that it can be compared in the study. But with more are more software being written every day, there might just be enough software out there that actually fulfills our requirements already. If we can get a hand on those programs and measure the efficiency of their development we just might be able to decide between C++ and Haskell feasibly. It will require even more statistics - the software analyzed will have not identical, nor even similar specification, it will be implemented by teams of various sizes, from various backgrounds, perhaps not entirely in the languages we are interested in. There will be unknowns (such as the actual development time) that we will have to work around. But if our sample sizes will be large enough, it might just be possible to refine the results enough for a statistically significant answer. To summarize, for our new approach, we require:

- access to very large amount of software programs
- knowledge of their development teams and other characteristics so that at least some control can be asserted

[2] https://www.statista.com/statistics/1127190/programming-languages-associated-highest-salaries-worldwide/

- viable software quality metrics that can be inferred from the data available
- statistical framework to correctly interpret the results

Due to the very large number of programs analyzed, it is imperative that the whole process is fully automated so that it can be processed in a practical time frame. And finally, as scientists, we should make sure that our method allows reproducibility so that our results can be verified and built upon by the research community. Recent evolution and widespread adoption of software development tools combined with the upsurge in the open source development popularity even among big companies have dramatically changed our ability to obtain and analyze large software repositories and the development process that led to their creation:

Version control systems, first introduced in [14] allowed programmers to record and track changes to the source code making progress on long duration projects easier. Version control systems quickly matured to *distributed version control systems* offering the same functionality for entire teams of developers collaborating on a single project. Their use skyrocketed around the time `git`, a decentralized distributed version control system conceived by Linus Torvalds for the Linux kernel developers has been published in 2005. The version control systems allow programmers to store smaller changes to the program in batches called commits. Each commit usually contains a text message explaining the change and the changes to the source code themselves. The version control system imposes order on commits made by different users, helps dealing with conflicts (i.e. when two developer alter the same part of the program at the same time) and allow reverting the program to any previous state.

Hosted Software Repositories With the increased availability of internet bandwidth and the transition towards cloud-based services several companies and organizations started providing internet hosting for version control systems around the turn of the century.

Open Source Software Development Since the beginning, some of the project hosting providers, such as SourceForge, specialized on open source projects, i.e. not only was the project hosted, but the source code was also accessible for all to download. Industrial providers, such as GitHub often offered a paid tier with private hosting and free public hosting for open source projects as well. As the providers increased their portfolio of hosted services to cover other aspects of the software development process, such as bug trackers, continous integration, release management and feature discussions, those were often provided for free to public open source projects as well, making the whole software development process open.

Open Source Adoption Finally, open source software has gained a lot of popularity in recent years. Even large companies often open source vast parts of their codebases and develop them in the open. One of the best examples of the amplitude of this shift is Microsoft: A company well known for fiercely protecting its codebase started using public GitHub for some of its projects as early as 2012. Over the years Microsoft become one of the largest open-source contributing companies on GitHub, including large software pieces such as Visual Studio Code, JavaScript engine for the original Edge browser and the Windows

Terminal, leading eventually to Microsoft buying GitHub in 2018. Microsoft is not alone in adopting open software and development principles. Many of the technology giants, including Google, Facebook and Apple as well as countless smaller companies worldwide have large public repositories available.

So here we are: We now have access to source code of millions of projects hosted publicly online. Better still those project vary from single person projects of passion to large open source applications and to applications developed in the industry. Thanks to the version control systems we can reconstruct the historic record of how the projects were developed and features added. And thanks to the public hosting of associated services, we can correlate the source code changes to new features, bugs and fixes using the metadata from issue metadata and continous integration logs, We can inspect the behavior of the development teams with relation to product releases via the release metadata and so on. The information about the entire software development process for millions of projects is at our fingertips and we can mine it to reliably analyze the effects our tools and abstractions have on it.

It should therefore be not surprising that as soon as those trends converged, researchers quickly recognized the potential of analyzing large software repositories and the method has been used in so many papers that it even gave rise to a specialized conference, Mining Software Repositories, available since 2004. While the initial research analyzing software repositories was merely setting the stage, operating on smaller datasets and exploring the boundaries of what can be analyzed, finally, in 2014 a paper titled *A Large Scale Study of Programming Languages and Code Quality in Github* appeared[13].

The paper attempts to answer the motivation question we set out earlier by analyzing about 1000 projects from Github in 17 different languages . Instead of focusing on the developer productivity in terms of the speed of development, which is next to impossible to determine from the available data, the paper analyzes another metric of development productivity - the number of errors in programs. For all other things being equal, if one language leads to fewer errors than the other, then its design is better as developers will be more productive simply by having less errors to fix. But how to automate finding errors in programs? Detecting the errors by analyzing the source code itself would be next to impossible so the paper turns to the history of the analyzed projects and analyzes the commit messages, short pieces of text that explain the change committed. When an error is spotted in an existing program, the error is usually corrected in a smallest possible commit and the commit message is very likely to reflect this information. Therefore instead of automatically detecting the programming errors, the paper uses the thousands of developers as classifiers and simply aggregates their findings.

So when the number of commits and the number of error-fixing commits is normalized for each project and language, the paper ends up with coefficients describing the propensity of each of the studied languages towards introducing errors in the programs. The paper reports those coefficients per language, aggregates the languages into groups of similar languages to judge the programming paradigms themselves and amongst other things,

includes more analyses to safeguard the results against uncontrolled variables, such as an exploration whether the ratio of errors rather than a language depends on the domain of the programs themselves, i.e. whether certain programming domains are simply harder than others and this extra complexity leads to more errors being observed instead of the language used (assuming that a languages correlate with program domains).

Overall, the paper has been very successful. As a hallmark of how statics and software repository mining can be used to answer the previosly impossible extremely complex questions, it has been reprinted as a research highlight three years later in the Communications of the ACM [12].

But, to quote a famous meme, one does not simply mine big data. While the scope and power of statistical analysis of big data is tempting, it comes with its own set of problems: The bigger the dataset the noisier it is. In the case of software repositories, noise comes from various sources. The version control system's functionality is advantageous not only to source code, but to a wide variety of tasks that store their information predominantly in a textual representation. Therefore software repositories such as GitHub consist of not just programs, but also books, examples, code snippets, programming tutorials, webpages, research papers (including this thesis) and so on. Furthermore, even when focusing on actual programs, the noise levels are huge. The projects are developed by people of varrying skill levels and teams of varying sizes. Majority of programming projects are short-lived and abandoned personal projects or student assignments. These projects are created by people often in the process of acquiring the skills in a particular languge and they are much less likely to follow the software development discipline.

For analyses such as the language quality paper outlined above, this is bad news. Results obtained from student projects, will likely not generalize to experienced developers. Furthermore, using projects that are not properly developed may even invalidate the methods of the paper: Detecting bugs by analyzing commit messages, a crucial step for the paper only works if the commit messages actually reflect those. This is true for large software projects with active user base and development teams, but much less likely in personal or even student projects.

Large software repositories are also full of copies of both actual code, and of entire histories of programs. Apart from the practice of copying verbatim source code snippets or entire libraries to projects that use them, Github allows an user to copy any project with its entire history into own for either collaboration, or customization purposes. Such project copies are called *forks* and they comprise up to a half of all projects hosted on the platform. While forks may also contain new code, including them in any analysis is dangerous as they may lead to overrepresentation.

To mitigate those issues, datasets obtained through large software repositories (same as any big data source) must be thoroughly filtered and cleaned before actual analysis otherwise they risk invalidation of their results on more rigorously curated datasets. This warning is not hypothetical, the paper on which we demonstrated the benefits of large software repositories suffered exactly this fate in [2] and [**?**].

Let us consider how a good data filtering for our demonstration paper would look like. The projects should be predominantly in the languages we are interested in (so that the the bug attribution to language would work) and should actually be rigorously developed to increase the chance of spotting bugfixed in commit messages. We should obviously exclude non-software projects, clones and forks as well. To put our requirements into more precise terms we may be looking for projects that fulfill the following criteria:

- projects whose majority of changes is in one of the languages we are interested in
- projects with more than 1 developer
- projects with more than 6 months of development (distance between the time of first and last commit) and at least 100 commits
- non forked and non-cloned projects

After applying those criteria we will still be left with many thousands, if not millions of projects and it would be infeasible to analyze them all. A sufficiently large random sample for each language should then be selected and analyzed. Finally, as researchers, we would like the process to be repeatable and modifyable in the future for easy reproducibility so that the paper can be easily checked against other languages, same languages at different times, or the filter criteria altered.

Unfortunately, doing so is currently impossible. Github and other large software repositories are tailored towards the use of one's own repositories. Searching for repositories is possible, the search queries are nowhere close to even the simple filter criteria expressed above. Search results also cannot be randomized, a necessary condition for any later processing. Any queries to the Github API are subjected to strict rate limiting so that even simple tasks such as project discovery (just learning the URLs from which projects can be cloned) is extremely time consuming task measured in months.

The only way to accomplish the filter as described above currently is to discover all github projects (month of work), then clone every project found. Once cloned, verify the other attributes and keep the project for further analysis if pass, or delete upon failure [3]. This operation alone would take many months and any slight change to the initial conditions would require a restart. Even worse with such a long timeframe, the dataset analyzed in the end will not correspond to any particular time - some projects will be analyzed in states months older than others.

Simply put, mining software repositories for precisely selected projects in reasonable amounts of time and in a reproducible manner is impossible. This lack of functionality is what drives many researchers to easily obtainable but vastly inferior substitutes, such as utilizing project popularity (one of the very few project properties Github allows semi-deterministic searches for).

And this lack of functionality is also the motivation for this thesis.

[3] Modulo the details of detecting duplicates and random sampling for which we ideally need the entire population

1.2 Thesis

Thesis statement: It is possible to create infrastructure that allows precise, scalable, deterministic and reproducible filtering of projects from large software repositories.

Such an infrastructure must improve over the state of the art in the following ways:

Data Acquisition and Querying Although there are multiple sufficiently large software repositories available, none of them provides enough expressivity and/or bandwidth to support reliable and targetted data acquisition without introducing any bias. Github for instance provides API for downloading entire projects, but projects discovery capabilities, i.e. determining which projects to actually download is extremely limited. Imagine we want to analyze projects that are written in the Java programming language and have been developed for at least 1 year, a reasonable expectation. Github offers no way to find those projects, let alone to obtain a random sample of those. Lacking the ability to explicitly describe the projects of interest, most of current research focuses on the undefined *developed projects* that are themselves selected by proxy measures such as project popularity. Data cleaning and filtering is often done ad-hoc with little understanding of the full contents are rsults are often left unverified.

Reproducibility Last, but not least, the volume of the collected data and the complexity of the selection makes reproducibility challenging. Obvious solutions, although being implemented by some existing papers do not work: Having all research papers analyze the same sample of projects is wrong as it means that whatever bias could have been in the selection will be repeated by every paper that uses it. And requiring each paper to archive with it the entire input dataset is (a) not feasible and (b) not enough: having only the analyzed dataset available does not allow checking that the dataset itself is robust.

The thesis introducesfour research papers that together advance the theoretical understanding and practical usability of large software repositories:

1. Analysis of cloning and associated biases in large collections of software repositories (paper 1)

2. Analysis of reproducibility issues and statistical interpretation of large corpora. Proposal of better methodology for reproducibility (paper 2)

3. Design and implementation of a tool for large scale download, archival and querying of software repositories to aid reproducble project selection and analysis (pape 3)

4. Analysis of the selection bias introduced by the most frequentlty used project selection by popularity convenience sampling on recent papers. Analysis of the obtained and missed projects and development of methodology for reproducible and validable project selection and associated tooling (paper 4)

1.3 Structure of the Dissertation Thesis

1. This chapter, the *Introduction* describes the motivation behind the thesis and introduces its goals.
2. *Background and State of the Art* surveys the past and current solutions and methodologies available for mining software repositories in order to analyze programming languages and software development practices.
3. *Overview of the Approach* summarizes the author's work towards more scalable and reproducible large software repository mining.
4. *Relevant Papers* presents a collection of author's related papers. These papers form the basis of the thesis and detail the contributions. A context and timeline for the papers is provided and each paper contains a detailed list of author's contributions.
5. *Conclusions* summarize the results of the thesis, hints at possible improvements and future work and concludes the thesis.

Chapter **2**

Background and State-of-the-Art

The benefits of analyzing large software repositories have been exploited for many years. During that time many tools have been either (ab)used or created especially for data mining. Recently, researchers have started to point out problems associated with large repository mining and advocated for an improved methodology. This chapter gives the overview of current and past work in these areas.

This section gives an overview of the main large software repositories and their basic properties. At its heart, a software repository comprises of two essential features: (a) a data source containing the projects and associated metadata and (b) an interface allowing retrieval and querying of the dataset. The size and composition of the data source define the theoretical usefulness of the repository, while the precision and speed of the retrieval determine the practical usability.

As an example, consider a repository that contains all of the software projects ever created. In theory this repository contains all of the inormation ever to be mined. However, if such repository provides only a very primitive API that allows retrieval of the projects from the oldest to the youngest that is limited by 1000 projects downloaded per day, the repository is practically unusable for anything but the analysis of the oldest projects as getting to the more recent projects would take decades. In other words, two key questions must be answered about a repository: What is in it and how do I get it.

Creating and maintaining a large database of software projects is a complex undertaking. Therefore, there exists only a relatively small number of such data sources and they are usually not built to primarily support a niche task like data mining, but focus on aiding the software development process itself. A larger number of softare repositories follow a different approach and instead of maintaining their own unique source, to reduce the costs, they choose to mirror an existing source, or its portion (in both size and kinds of data stored) and often provide more complex interface to data retrieval that is better suited for data mining.

This chapter splits the discussion about existing software repositories into first discussing the primary data sources available in terms of their composition and volume. It then

looks at the software repositories from the querying and retrieval perspective alone. A software repository that also maintains its own data source thus appears in both sections.

2.1 Sources

A software source can be characterized by the following main attributes:

- *Size and bias* - the number of projects available in the store and any bias associtated with the store. For all the sources mentioned in this section, one has to accept the bias towards publicly available open source software, but other biases, such as programming language, project popularity and so on are mentioned where appropriate.
- *Contents* - data sources differ in the data they store. Historically the first sources we built around version control system hosting providers and therefore they contained not only the most recent code but also a history of changes as recorded by the VCS. As development switched to open distributed model, extra metadata information about the software, such as code reviews, regression test results and so on become available. Dedicated repositories may also synthesize and store extra software engineering metrics.

The rest of this section describes the main software sources in the above terms.

2.1.1 Github

GitHub started in 2008 as a online hosting for git projects. Its key advantage was the inclusion of a free plan that allowed individuals and companies to host unlimited number of open source projects with paid plans for private and closed projects. Over the years, this policy together with the popularity rise of git itself has made GitHub extremely popular amongst programmers.

This success has made GitHub the hegemon in terms of number of software projects stored and the variety of extra information available. GitHub hosts an enormous number of projects: As of 2022, there is over 230M of publicly available projects on GitHub and the total number of projects hosted might well attach half a billion (this number cannot be obtained precisely as private repositories cannot be distinguished from deleted ones). GitHub is used by over 73M developers [9].

GitHub's focus on supporting the development process has some important implications for its quality as a software repository source: As its popularity increased, GitHub diversified the services provided and went from a simple version control system hosting to a complete platform for software development, integrating a plethora of extra services under its name. GitHub now provides project web hosting, issue tracking, code review process, continuous integration builds, release management, deployment and much more. This vast portfolio of extra services and their widespread use as most are still offered free of charge for open source projects has made GitHub a treasure trove of both software projects' code and

an extensive information of their entire development process and deployment and usage patterns.

All this makes GitHub the single most complete source currently available, but this popularity comes at a price, especially when mining software repositories is the task at hand: The notoriety of GitHub and the amount of extra services it sports has led its use to transcend the original software development niche. Large amount of projects hosted on GitHub are not software per se (including, but not limited to hosted web pages, book manuscripts and documentation). Even larger amounts are pieces of software that was never intended to be developed as GitHub is a popular vehicle for conducting computer science courses (with repositories automatically generated in vast numbers for all enrolled students), showcasing demo applications and even a dump site for abandoned projects for archival purposes.

Finally, being oriented towards the development process itself, GitHub does not provide any guarantees about future ability of its data. Projects can be deleted or made private, their histories can be altered, or even purged and none of these changes are archived. The new content, or lack thereof simply overwrites the past data with no going back.

2.1.2 Bitbucket

Bitbucket is the other relevant primary software repository. Founded in 2008 as a hosting service for Mercurial, another version control system, following the upsurge in git's popularity, git was added as option to bitbucket as well and finally, Mercurial support was removed from Bitbucket in 2020, cementing git's dominance. Bitbucket is much smaller and thanks to its policy of allowing privately hosted repositories for free, unlike Github it is used much less for open source software. In terms of repository contents, Bitbucket is very similar to github, offering own bug trackers, discussions, pull requests and continous integration.

As Bitbucket is much less oriented towards supporting the open source community, it is much harder to determine its size. Furthermore, as Bitbucket uses unique text identifiers for its projects, simple enumeration as in the case of GitHub is also not possible. A reasonable estimate of 3.4M public projects can be obtained from archival sites records[1].

2.1.3 Other VCS Hosts

Numerous other primarily version control system based software sources exist. As their usefulnes for software repository mining pales with the comparison of GitHub, they are only briefly discussed in the following paragraphs:

GitLab GitLab is a service bearing striking similarity to GitHub itself with a major twist: GitLab is geared towards a self-hosted deployment, making its usefulness as a project

[1] Number of Bitbucket projects in Software Heritage corpus is about 2M projects, compared to 136M for GitHub. Using the same ratio of completeness for both providers would give us 3.4M public projects

source more complicated as large amount of GitLab instances would have to be scanned for reasonable number of projects to be acquired. In addition to self-hosted option, GitLab also provides cloud hosting, with the number of projects available reaching 3.4M/footnoteVia Software Heritage.

SourceForge Created to support open software project and their development directly with no paid options, SourceForge provides capabilities similar to alredy mentioned services. Its membership stands at over 500K projects[2]. In addition to the popular git VCS, SourceForge supports also Mercurial, CVS and others.

2.1.4 Package Managers

Package managers provide software developers with access to large numbers of third party libraries available for their programs that can be easily integrated into applications. Instead of the continuous development process supported by version control systems, package managers focus on the releases - updates to the libraries they save made explicit by the users.

Package managers contain less noise in the form of non-software projects and often provide extra metadata about the usage and downloads analysis of the packages that can be used to further filter the interesting projects. Historic reproducibility is also better than version control systems as older versions of packages are kept for backwards compatibility. Unfortunately, a package can still be withdrawn by its developers, such as the infamous withdrawal of leftpad in 2016[3].

In terms of size, package managers for the most popular platforms, such as JavaScript reach millions of libraries, while less widespread languages such as the R programming language usef mainly for statistics with its CRAN package manager with merely 20k packages.

Their biggest weakness is strong bias towards library code as virtually no applications (standalone executables) are part of any package manager. And while this may not be a problem for library code oriented analyses, package managers are rarely the primary source of the code and most of their contents is available from version control systems, notably GitHub.

2.1.5 Software Heritage

Started in 2015 by Inria, the Software Heritage Project [4] aims to preserve the large code base mankind has created. It archives software projects from various primary sources including GitHub, Bitbucket, Gitlab, CRAN, SourceForge and Debian repositories. The project is actively maintained and updated via means of automated crawlers or direct access by partners. As of Fall 2022 the Software Heritage has archived over 184M projects with GitHub being its major source with 136m projects, followed from a distance with

[2] `https://sourceforge.net/about`

[3] `https://www.theregister.com/2016/03/23/npm_left_pad_chaos/`

GitLab (4M), Bitbucket (2M) and NPM (1.8M). All other sources contribute less than 1m projects.Compared to development oriented platforms such as GitHub or BitBucket, Software Heritage stores only limited metadata directly conncted to the software itself, such as commit messages.

Similarly to package managers, Software Heritage is not the primary source of the programs it stores and merely archives their code. But since its mission is the archival of the source, for the purposes of this thesis we count it as a source as well. As such, it is the only source providing excellent historical reproducibility which is provided by snapshoting the software - every time a crawler hits upon a software project already known to the archive, a new snapshot is created. Therefore noting the unique identifier of a snapshot used allows accessing the same data in the future and no data is lost.

2.2 Software Repositories

After describing the sources themselves, this section focuses on the software repositories themselves. Similarly to a source, software repositories can be characterized by a few key attributes:

- Sources - whether a repository maintains its own primary source, creates a mirror of its own, or simply provides a frontend to another repository.

- Active - repositories are accessible and can be used. Inactive repositories are included for their historical signifficance. *TODO You said in the notes this should be removed, but I feel just having the sentence about active ones is a bit too little in the newer version. If that's not the case, please mark again*

- *Updated* - a repository can be accessible, but may hold stale data. Some repositories offer a single view of the projects they store, while others are regularly updated at varying intervals.

- *Query power* - while virtually any repository provides *some* form of querying the projects it contains, the expresiveness of the queries is a limiting factor. For these purposes, a *basic* querying capability indicates that querying via few selected attributes is supported (such as popularity, language, size, etc.). *Filter* query indicates that the query attributes can be combined to form more complex filters. *Full* querying capacity allows queries to execute over all items stored in the repository, usually using a query language, such as SQL.

- *Deterministic* - a deterministic repository will, for a given query, return always the same answer as long as its underlying data remains unchanged. Determinism of a repository is usually violated by advanced distribution techniques, such as load ballancing.

- *Reproducible* - a reproducible repository goes beyond simple determinism by requiring that a query can be constructed in such way that identical results are returned even if the underlying data gets updated in the meantime. Non-updating repositories achieve reproducibility trivially, for updationg repositories, especially the primary ones, reproducibility is much harder as projects may be deleted or their history altered using version control systems.

This section describes the major software repositories along those attributes, paying attention to any limitations. To illustrate these, we describe how each of the repositories can be used to obtain a random sample of 10000 Haskell and C++ projects with at least 50 commits together with their commit messages and issues for the analysis described in the introduction chapter. *TODO this is my attempt to make the sections that deal with the usage details to be more part of the thesis. I hope it works*

2.2.1 GitHub

Although primarily a data source, GitHub is also a software repository. Very much active in 2022 and thanks to its main purpose of VCS hosting, GitHub is constantly updated as projects are being updated, or indeed created (TODO fill in rate). However, those project can also be deleted, made private, or their histories may be overwritten by their authors. Since GitHub does not keep historic records, it is not reproducible. GitHub offers multiple different ways of accessing the data it stores and while all of those have been used for data mining purposes previously, none of the querying mechanisms have been created for that purpose. The next paragraphs details the various APIs, their original purpose and discusses their shortcomings for the purposes of this thesis:

Git The simplest method is to simply use git, the underlying version control system, to retrieve the contents and history of hosted projects. The git access exists so that developers can obtain the project (one at a time) they are involved with and upload their changes. It therefore features no project filtering, or even discovery capabilities so the urls of the projects downloaded must be known by some other means. Furthermore, only information maintained by git itself is accessible using this method (only file contents and commit history). GitHub imposes no official data rate limits on downloading repository contents via the git API, but we have observed heavy throttling for continous multi-thread accesses. Despite the throttling, cloning GitHub projects remains the fastest method of getting the actual contents en masse. The git access is deterministic.

REST API Provides the complete access to all data types available on GitHub. The API is geared towards programmatic inspection and manipulation of owned projects, such as automatic releasing, code scanning, pull request alerts and summaries, etc. The API also provides an endpoint capable of searching for projects based on simple queries that allow filtering based on user or organization name, project description and readme contents,

project size in bytes, number of followers, forks, stars, creation and last update time, programming language used, topics associated with the project, issues ready for contribution and various project properties (mirrors, forks, archived projects, sponsorability). Limited sorting is supported (by stars, forks and help-wanted issues). Looking at the search capabilities, its intended use is to promote community involvement, not any form of mining. This is exacerbated by the limitation of at most 1000 search results per query, which makes constructing larger datasets an impossibility. A query must not be longer than 256 characters and can contain at most 5 clauses. The query results are not guaranteed to be deterministic - if a query timeouts, partial results found so far are returned. Special endpoint exists that allows listing all public projects in the order of their creation with no additional filtering capabilities. Unlike the other queries, this one is not limited to 1000 results and can therefore be used to obtain urls of all public projects hosted on GitHub. The REST API has rate limits that reduce its appeal for mining purposes. At most 5000 requests per hour can be made by a single user, whereas the search API incurrs additional rate limit at no more than 30 search requests per minute. GitHub also uses secondary rate limiting which may be activated at any time GitHub suspects overuse of its resources, details of which are not publicly disclosed.

GraphQL API On top of the REST API, GitHub provides a GraphQL API. This API offers more precise queries to be formed, but its main advantage is the precise control over the data returned and the ability to construct a single query that would have required multiple REST API endpoints. For instance, to return all mergeable pull requsts of a repository, the REST API would need a call to determine pull-requests of a repository first and then a call per pull request to determine whether it is mergeable. The GraphQL API can achieve the same result in a single query. For the purposes of data mining though, it suffers from the same drawbacks: limited query power, severe rate limiting, inability to fetch *all* results (GraphQL queries are limited to 500000 nodes, the meaning of a node depends on the exact query) and non-determinism.

Web search Not indended for automated use, the web search presents itself as a search bar within Github's webpage. On top of searching for repositories via same queries as the REST API, the web search also allows searching for particular files and even pattern matching over the file contents. Web search results are non-deterministic even for simple queries that do not timeout, presumably due to load ballancing of the web requests.

Although none of the provided querying mechanisms is expressive enough to retrieve the projects we are interested in our example, combining them together takes us closer: The search API must be ruled out due to the limitation of 1000 results. We must therefore use the public repositories query and obtain all of the public projects. At 100 projects returned per request and given GitHub's size of 230M public projects and the rate limiting, this step would take 19 days to complete and give us the urls of all public projects. We then must determine those that belong to Haskell and C++, which at a single request per project

would cost us approximately 5 years. We can parallelize this step at the cost of risking GitHub's wrath, or select urls at random until we accumulate enough in both languages. This would be relatively straightforward for C++, but given to its relative scarcity might be impractical for Haskell. We would then need an extra request per project to determine the number of commits. Once we have the urls of projects that belong to the sample we are interested in, we can proceed to download their contents using the git API and the information about their issues via the REST API (there would be additional requests here, but the subset is now small). Alternatively, we can get faster results downloading all the public projects in parallel, then analyzing the sources to determine language and number of commits, keeping only the projects we are interested in. Assuming a 30 second download per project, 100 downloads in parallel, the whole process can be completed in about 2 years.

Those timelines are squarely outside of the realm of practicality and they no-doubt provide a worst case scenario. Clever hacks can be used to speed up the various parts, we can use other repositories with their own GitHub mirrors (such as GHTorrent described below) to speed some parts of the process. But despite all those effors and the tradeoffs we would make along the road, it would still take a lot of effort to write all the code gathering the information from various sources and APIs and the data will be ready in moths at least.

TODO Should I mention here that this is why people often fall to do the top 1k stars, which is a query they can get instantly. SHould I point out how unreproducible this would be?

2.2.2 Software Heritage

Although Software Heritage is updated both for new software and for new updates to existing projects, the update and discovery rates are slow and unpredictable and can go into years[4]. Softare Heritage is both deterministic and reproducible as different visits for the same project are all archived and can be retrieved separately.

As Software Heritage stores projects from various primary sources, it offers a special API, called *Vault* that can asynchronously collect any archived project and export it as a bundle in a variety of formats, such as `git`. In this regard the vault service is essentially equivalent to downloading projects from GitHub.

Each piece of software is assigned an unique identifier and this identifier, project name, url and and assigned tags are the only things Software Heritage can query on. The API is limited to 1000 results per request. It offers a REST API similar to that of GitHub with endpoints geared towards retrieval of known items, not advanced search and filtering. Rate limiting is more severe at 1200 requests per hour per authenticated user. Software Heritage is work in progress and it is likely the querying capabilities will increae in the future.

[4]As an anecdotal evidence, this thesis author's own software *terminalpp* has been discovered and archived, but not updated in two years. Software Heritage offers manual trigger for selected projects in such cases, but this approach does not scale

Using Software Heritage for our example would therefore suffer from the same weaknesses GitHub did, namely severely limited querying capabilities and rate limits. Furthermore, as Software Heritage only captures the code and its history, we would not be able to obtain the issues and would have to use GitHub or other primary sources for them, making data acquisition much more complex and loosing reproducibility in the process.

TODO this is a research project and likely if we asked, we would have been given better access to it as established researchers. Should we mention it?

2.2.3 GH Torrent

GhTorrent started in 2012 as an scalable, queriable and offline mirror to data offered through the GitHub APIs. It monitors the GitHub public timeline, a special API endpoint that publishes many GitHub public events in a single stream. The stream allows the GhTorrent crawler to observe each such event in real-time and store them in its own offline database. There have been no updates to the project since 2020. At its peak, GHTorrent archived over 18 TB (compressed) of recorded activity for more than 150M projects. For selected projects, their activity outside of the GHTorrent's active years was analyzed and added to the database.

The database exists in two versions, a MongoDB dump of the raw GitHub public event timeline records and a SQL version that contains the information processed from the public events, such as basic project information, popularity, commits, messages and comments, issues and so on. Almost every metainformation available on GitHub is archived and processed with the notable exception of actual file contents. Both MongoDB and SQL databases could be searched online, and can still be downloaded offline for local use. Rate limits are moot in the local download scenario and determinism and reproducibility were guaranteed via the monthly released snapshots.

The databases also form the querying and retrieval API for the repository. All of the archived information can be can be searched, filtered and ordered easily by complex queries that far surpass the ability of GitHub or Software Heritage. Even more complex queries can be calculated offline as GHTorrent archives enough metadata to allow calculation of various aggregated software engineering metrics.

GitHub suffers from two major drawbacks (other than not being active anymore): the dataset integrity is not guaranteed. Due to the nature of its data accretion, it favours active projects. Downtime in either the crawler, of GitHub public events timeline results in data loss for GHTorrent. Efforts have been made to remedy some of those by recrawling, but the dataset is filled with inconsistencies (textit TODO quote our last paper with the info about the big dip). Furthermore, the dataset only consists of the metadata published on the timeline and most notably lacks any source code.

Due to the lack of source code, GHTorrent alone cannot be used for our example. But when used together with GitHub it greatly speeds up the process: The latest snapshot of GHTorrent can be downloaded and then queried for Haskell and C++ projects with sufficient amount of commits in a matter of mere hours. Since GitHub public events

timeline also contains information about issues, these too can be obtained from GHTorrent and virtually no additional costs (but likely with errors). The urls obtained can then be used to download the projects' source code directly from GitHub. Since the GHTorrent dataset contains errors, some of the selected projects will not be available while others will not fulfill the required criteria, but we can compensate by selecting more projects initially. However, by using GitHub we again lose reproducibility. *TODO should I give real examples here, like when we used GHT and GH for the TOPLAS to reconstruct the dataset, etc*

2.2.4 Orion

Orion [3] is an example of a different approach to mining software repositories. Instea of simply maintaining large dataset and/or providing access to one, it recognizes the difficulty in using the raw databases and focuses on scalable and expressive querying mechanisms. Its dataset comprises of 185K projects from GitHub, Google Code and SourceForge and contains all kinds of information from the project contents, version control history, metadata and synthesized attributes related to software engineering. The project is no longer maintained.

The design of a domain specific language for querying large software repositories and the implementation of its search engine is the main focus of the paper. To showcast the querying capabilities, the paper even opens with a sample question: *"What projects contain more than 10,000 lines of code developed by less than 10 people and are still actively maintained with a high bug-fixing rate?"*. The queries link different artifacts of software development, such as the code itself, version control history, addition metadata and synthesized arguments with software enginnering signifficance.

Orion is both consistent and reproducible trivially as the database has not been updated. Live synchronization has not been part of the project. If updated, it would provide similar guarantees to GHTorrent, i.e. periodic snapshots of the database.

Obtaining the required projects via Orion would have been simple had it been still available. However as Orion did not support updates to the corpus, the data we would get would be extremely outdated now. Furthermore, for larger studies, it has not been demonstrated that Orion's approach would scale beyond hundreds of thousands of projects.

2.2.5 Boa

Like Orion, Boa [5] addresses the need of efficient searching over large software repositories. Boa goes even further and strives to provide tools to mine specifically the source code itself. The dataset started with 490K Java projects obtained from SourceForge, but later switched to GitHub (380K projects since 2015) and was extended to support Python (2020) and Kotlin (2021). The dataset contains additional 7.5M Java projects for which the source code information is not parsed and therefore Boa would only search for the aggregate project attributes. The dataset is updated and snapshoted infrequently with no changes

to the Java dataset since 2015. Boa is both deterministic and reproducible due to the infrequent dataset updates.

Boa supports not only serching the aggregated attributes, but also parsed abstract syntax trees of the stored source files. It then uses own domain specific language based on the visitor pattern to allow constructing efficient queries over the syntax trees and project attrributes. It thus offers the biggest expressing power from the tools reviewed. The queries are excuted in parallel on a hadoop cluster and provide fast retrieval.

Access to the Boa dataset must be requested first, after which the full potential of the system is available to its users including the computing infrastructure to run the queries on.

Although in terms of the querying capability, Boa is the most complete solution thans to its ability to search the code as well, this comes at a price: Adding new language to Boa is a substantial effort and unfortunately, neither C++, nor Haskell are supported as of 2022. Furthermore, the infrequent updates would make the results quickly obsolete (it's Java corpus is now 7 years old). It also remains to be seen if Boa can scale further.

2.2.6 Other Repositories

This section provides a cursory overview of other software repositories. While some of them are still active, they are of much smaller scope in either dataset size or features and as such cannot be used for the purposes of this thesis. We mention them for their historic value and/or uniqueness of their approach.

Bitbucket The similarity with GitHub as a source of data also translates to Bitbucket as a software repository. Although Bitbucket provides no web search facility, its REST API is very similar to that of GitHub. Its querying capabilities are slightly more advanced, but the amount of searchable attributes is still far from being useful for efficient mining. One notable improvement is that anything than can be filtered can also be sorted. However, randomization of results is not supported and due to the use of text unique identifiers for projects, random project acquisition is not possible either.

Flossmetrics This work analyzed 2800 open source projects and computed statistics about various aspects of their development process, such as number of commits and developers [7]. Information from additional sources such as project mailing lists and issue trackers was included. Queries could be formulated on metrics such as COCOMO effort, core team members, evolution and dynamics of bugs. Filtering based on these criteria was supported. The project is inactive and it did not support updates.

Black Duck Open Hub A public directory of open source software[5] that offers search services for discovering, evaluating, tracking, and comparing projects. It bears similarity

[5] `https://www.openhub.net`

to the older Flossmetrics projects upon which it improves in both quality and quantity, including continous updates. It analyzes both the code's history and ongoing updates to provide reports about the composition and activity of code bases. The Open hub does not store any contents of the analyzed projects, nor does it keep historic data other than the aggregated metrics. However since the links between Open Hub projects and their repositories is kept and the querying capabilities over the analyzed attributes are extensible, the Open Hub can be used to bootstrap an analysis by selecting projects whose contents will be downloaded from a primary source. Open Hub does nort suppoort randomization of the results, but given its relatively small size, getting all the data first and then doing own randomization is indeed a possibility.

Sourcerer The single aim of this project is to detect code clones [15]. The tool scales to large datasets and can detect near-identical code at various granularities. It has been used to analyze cloning across large corpora of Java, JavaScript, Python, C and C++ projects on GitHub [**?**]. Sourcerer does not keep the metadata or code of the analyzed projects, but keeps a hash of each file contents as well as a fingerprint obtained by tokenizing each file and remembering the token counts. Only source files in the four analyzed languages are kept. It could be used by researchers to detect duplication in their samples specified by links to GitHub projects, after which a report of cloned files found within the dataset was provided. The project's web page appears to be inactive.

Stress [6] One of the first attempts at reproducibility of project selection, stress works either locally, or online (with the url now defunct). Its accompanying paper surveys the reproducibility of project selections in 68 studies and finds none to be completely reproducible. It then proposes a selection tool that allows extensive filtering based on the project information and 100 synthesized arguments from the projects version control data and metadata, such as project lifetime, open tickets, etc.. The tool is verified on a corpus of 211 Apache projects. Stress supports queries to be stored and repeated later. Querying over source code is not supported.

2.3 Summary

This chapter reviewed current and past software repositories and their data sources in terms of their size, querying capacity and reproducibility, all of which are essential for their mining for research purposes, summary of which is presented in table 2.1.

It shows that there indeed exist software repositories that are either (a) large, (b) support access to source code, (c) offer expressive filtering capabilities that are (d) deterministic and (e) reproducible and (f) are up-to date with current development of the projects they contain. However, integrating those features into a single repository is challenging and none of the existing solutions provide all of the features together.

	Size	Sources	Active	Updates	Deterministic	Reproducible	Query	Contents
GitHub	210M	primary	Y	continuous	–	basic	code,vcs,meta	
Software Heritage [4]	175M	many	Y	continuous	Y	basic	code,vcs	
GHTorrent [8]	157M	GitHub	–	continuous	–	full	vcs,meta	
Orion [3]	185K	many	–	–	Y	Y	full	code,vcs,meta,attr
Boa [5]	980K	Github	Y	–	Y	Y	full	code,vcs,meta,attr
	8M	Github	Y	regular	Y	full	meta,attr	
Bitbucket		primary	Y	continuous	–	basic	code,vcs,meta	
Flossmetrics [7]	2800	many	–	–	Y	Y	filter	attrs
Black Duck	1.4M	many	Y	continuous	–	filter	attrs	
Stress [6]	211	Apache	–	–	Y	Y	full	attrs
Sourcerer [15]	4.5M	GitHub	–	–	Y	Y	basic	attrs

Table 2.1: Comparison of major data sources and associated tooling. Repositories are classified in terms of their size, sources (primary menaning the repository maintains its own data store), whether they are still active, if and how a repository supports updates (not at all, regular dumps, or continuous), whether its queries are deterministic and reproducible

, the power of queries (basic pre-set queries, filter over multiple attributes and full querying of all stored data) and the types of data stored per project (general project attributes only, version control information (commit history), additional metadata (such as issues, etc.) and actual code).

Chapter 3

Overview of Our Approach

Many of the existing tools and datasets reviewed in the previous chapter state the prohibitive cost of large repository analysis as their motivation [3, 5]. Ddespite their advancements, it has indeed been my experience as well. My original background is from the programming languages, not data mining and I wanted to analyze the modern JavaScript language and its evolution. But before I could even attempt a moderate analysis, enormous amount of work had to be done. This work was not in vain as it both generated interesting results of its own [10] and identified the ongoing need of better tooling for the very large repositories we have access to. The lack of tooling led to my switch from programming languages analysis to large software repository mining and to this thesis.

The existing approaches share a common theme: The larger the dataset, the fewer features. This is not surprising as any functionality becomes harder at scale. Corners must be cut and compromises must be made as workarounds. GHTorrent discards project contents and dataset integrity for scale and speed which limits its use for any more complicated analyses to simply non-reproducible project discovery. Software Heritage prioritizes long-time preservation over retrieval and short-term correspondence with reality. Orion and Boa sacrifice the same attachment to reality and scale in order to get much more powerful search capabilities. All those sacrifices make sense for certain applications, but I believe that for the specific purpose of supporting mining very large software repositories for mainly research purposes, they all fall short of the goal.

Let us recall the thesis statement, that aims to create an infrastructure that allows precise, scalable, deterministic and reproducible filtering of projects from large software repositories. This chapter introduces CodeDJ, the solution presented in this thesis, a queryable large software repository built with expresiveness, scalability, long-time relevance and reproducibility in mind.

3.1 Sources

The most important design consideration is what the dataset should actually consist of as it has profound impact on the other categories as well. We investigate the dataset

description from two angles: that of its primary sources, and the types of data stored in it.

Sources Not limiting oneself to a single primary source has its benefits, namely more coverage, which is why many of the tools from ten years ago opted for multiple sources. However, this is one of the very few categories where the development in the last decade has made the design actually easier: In 2022, GitHub is the only relevant primary data source. Software Heritage, which aims to preserve software from all kinds of sources is essentially an outdated GitHub mirror. Of the 180M projects it archives, 135M (74%) come from GitHub. The second most represented source is Gitlab with mere 4M projects. While it is possible that certain niche populations are not represented enough on GitHub, evidence suggests that such niche populations will be exceedingly rare as many other sources either point to GitHub projects in the first place, or simply provide a GitHub mirror for user convenience. In 2019 I have analyzed XYZ JavaScript NPM packages (a Node.js package manager and also a Software Heritage source with close to 2M projects) and found that XYZ% of them are actually developed as GitHub projects. This number is even greater for more popular packages. Similar situation exists in other popular repositories, such as CRAN for R language.

The dominance of a single source increases ever more when we consider not the provider, but the version control system itself. While different hosting services offer different extra products and their accompanied metadata (user information, discussions, etc.), the core history of the project's contents and its updates is the same and in the same format as long as the same VCS is used. Here `git` reigns supreme. Used exclusively by Software Heritage three biggest providers, GitHub, GitLab and Bitbucket and countless others, it is the de facto standard VCS of today.

The situation is more complex for non code related features, such as continous integration, issue tracking systems, project discussions, etc. While GitHub offers an in-house alternative to almost all of such services, the market is much more fragmented (BugZilla, JIRA, Jenkins, TFS, Travis, Appveyor, etc.)

Finally not all source code relevant for research and analysis is found in GitHub repositories. A lot of code comes from less organized and smaller form factors such as Jupyter and R Notebooks, documentation and examples and even on-line helper forums such as StackOverflow.

And although there is nothing that can be said about private repositories, it should be noted that such repositories are nowhere close to extinction in the wild. The hegemony of GitHub in the private ranks is not as obvious with services as Team Foundation Server and cetralized solutions such as Perforce or even previous generation of VCS such as Subversion and Current Versions System (cvs) seem to have much larger presence in the corporate sector. Even on GitHub, while exact data are scarce, we can observe that public projects form only 42% of all project identifiers know to GitHub[1].

[1] The remaining 58% is either deleted projects, or private projects and it is impossible to differentiate between the two.

Our extensive use of GHTorrent has made us aware of the severe drawbacks of using one source for project selection and the other for actual data for analysis. If either of the data sources is not reproducible (as is the case of GitHub as contents source), such use cases must give up on reproducibility. Furthermore extensive effort is often needed to make sure the two datasets are consistent, i.e. the contents obtained from second dataset correspond to the state of the first dataset used for their selection. Another argument for storing project's contents is that without the contents, the selection queries the dataset is capable of must exclude any search in files, a limiting factor especially for the programming languages research. A single dataset for both selection and contents is thus much easier to use, but equally harder to maintain as the volumes that need to be stored and processed will immediately be dominated by the stored contents.

We have thus decided to design CodeDJ around `git` as the sole provider for project contents and GitHub as the sole provider of the metadata.

3.2 GitHub Ecosystem

In order to better understand the expected volumes and composition of data that CodeDJ would have to hold, we have conducted various analyses of the GitHub ecosystem.

3.2.1 Duplication

In an attempt to estimate the volume of data CodeDJ would have to store we turned our attention first towards cloning. We analyzed projects in four different programming languages (JavaScript, Java, Python and C/C++) chosen for their different usage scenarios and software development practices to minimize chances of bias and analyzed file-level identical and nearly identical files. Our study found that the percentage of file-level clones is high, ranging from 40% for Java to a staggering 94% of all JavaScript files being copies. Details of the analysis can be found in [10].

We now present a more detailed study of the JavaScript ecosystem alone. The study not only sheds light on the types of cloning used, but since it was done before CodeDJ it also illustrates the costs of large software repository mining. Figure 3.1 shows the stages of the data acquisition and analysis pipeline. The pipeline is divided into three parts: data sources, data preparation, and analysis. Data sources are the raw data that we retrieved from various sources. Data preparation combines, filters, and cleans the data we initially retrieved to create a data source we can use for performing analysis. Analyses each perform an examination of some specific portion of the data, and then remove the data they analyzed, preparing the field for the next analysis in the sequence.

Data Sources Our initial data set comes from three sources. The bulk of the data we use comes from the *GHTorrent* data source, which was scraped from GitHub over a period of 4 mo by cloning repositories and extracting information about each repository using Git. Due to space constraints, we discarded each repository after scraping and

only keep the generated summary. The list of repositories to extract was compiled by selecting all of the JavaScript projects in GHTorrent and consists of 3.3M repositories. We limit our scraping efforts to just the master branch of each repository. The most basic data we extract for each repository consists of commit metadata which includes commit hashes (used to identify commits), author and committer emails (which identify persons resposnible for commits), author and commit timestamps (used to place commits on a timeline), and commit tags (to determine the master branch), as well as commit parent-child relationships. For each commit, we also extract a list of file changes. Each change is specified by a before and after hash, and either a single the file path for ordinary file modifiations, or old and new file paths in case of renaming. Finally we retrieve the history of the contents of each repository's submodule file which allows us to keep track of which directories are submodules throughout the life of each repository. This data is collected for a list of 3.3M projects and contains information about 62M commits and 2.1B changes.

NPM is a supplementary data source that is extracted in the same way as GHTorrent, except the list of projects is compiled from the addresses of GitHub repositories of NPM packages that advertise a them in their JSON manifests. This data source took 14 d to retrieve and holds data from 194K repositories and includes 10M commits and 110M changes.

GitHub metadata is the third data source. It is obtained via over 28 d days via the GitHub REST API by retrieving basic project information[2] for each of the 1.8M repositories found in either of the two other data sources.

Overall the data acquisition took over 5 months, excluding the time required to develop and debug the scrappers. We had to resort to cloud-based virtual machine providers for distributed acquisition to at least partially circumvent rate limiting induced by GitHub for project cloning. The metadata information was also distributed using multiple GitHub identities for the same reasons.

Data preparation After the acquisition, the data obtained from the three distinct sources had to be merged together and cleaned. This process took place in four consecutive steps:

This first step is to *join* the data coming in from the NPM and GHTorrent data sources into a single data set of 3.5M projects, 66M commits, and 2.1B changes. The join step takes 12 h to run and the result takes up 106GB on disk.

We then proceed to *patch commit creation times*. Since the timestamps used by `git` are arbitrary, a small number of projects contains child commits older than their parents. Since this makes reasoning about commit histories more complex, we patch commit times so that no commit is younger than its oldest child. This affected 1% of commits.

To determine original projects, we also needed to know when particular project was created. This data is not something we can gather form Git, but must get directly from GitHub (note that a project creation time is not identical to the time of its first commit). Thus in step three we turned to two sources: we extracted the timestamps from *GHTorrent*

[2] https://developer.github.com/v3/projects/#get-a-project

for the majority of projects and *GitHub metadata* for those not listed in *GHTorrent* and attached them to the data set. We managed to retrieve project creation times for 98% repositories and we discarded the remaining 2%. The discarded repositories accounted for 1% commits and 4% changes.

Compared to the data acquisition, data preparation took less than a day on a machine with 2 Intel Xeon Gold 6140 CPUs with total of 36 cores and 72 threads and 512GB memory.

Analysis Finally, the actual analysis was performed in five distinct steps. In each step we extract and analyze discrete a portion of the data set that comports to a specific type of cloning behavior. After each step of the analysis, we discard the just-scrutinized subset of the data from further analysis.

While our data set specifically precludes GitHub forks, not all forks are created through the GitHub interface, and therefore not all forks could have been properly filtered out yet. It is possible to fork a project by checking it out and manually changing its origin. This preserves the projects history intact but allows the project to diverge from the original at the point of cloning. The first step of the analysis scrutinizes the 7% of the remaining projects in our data set that constitute such manaually created forks which account for 5% of the remaining commits and 32% of the changes. We show the results of this part of the analysis are presented in Sec. **??**.

In the second and third steps of the analysis we look at project clones. Among these we distinguish two distinct cloning practices. Root-to-root project cloning acts as impromptu project forking. These implicit forks are created by copying all of the files in some repository to the root of a new repository. In this way, the implicit fork does not preserve the commit history of the original project. Such impromptu forks make up 4% of our data at this point in the pipeline and constitute 5% of all commits and 32% of all changes. The other class of project cloning is root-to-subfolder cloning. when this happens, the cloned project becomes an ersatz submodule in the cloning project. As opposed to actual submodules, an ersatz submodule recieves no special treatment, so it is up to the developer to keep them updated. They may, but might not, keep around the history of the original repository. Only ¡1% of the remaining projects contain such ersatz submodules, which make up ¡1% of commits and 2% of changes. This results of this part of the analysis are presented in Sec. **??**

In the fourth step we analyze folder clones, which occur when a folder at some point in the lifetime of some repository is copied to another repository. % of the remaining projects contain such clones and they affect % of commits and % of changes. We analyze folder clones in detail in Sec. **??**

The final step of the analysis is concerned with file clones, where individual files or groups of files are copied. They make up % of the data set at this point in the pipeline and affect % of the remaining commits and % of changes. We discuss these clones further in Sec. **??**

After we analyzed and removed the clones, we are left with 2.8M repositories, 52M

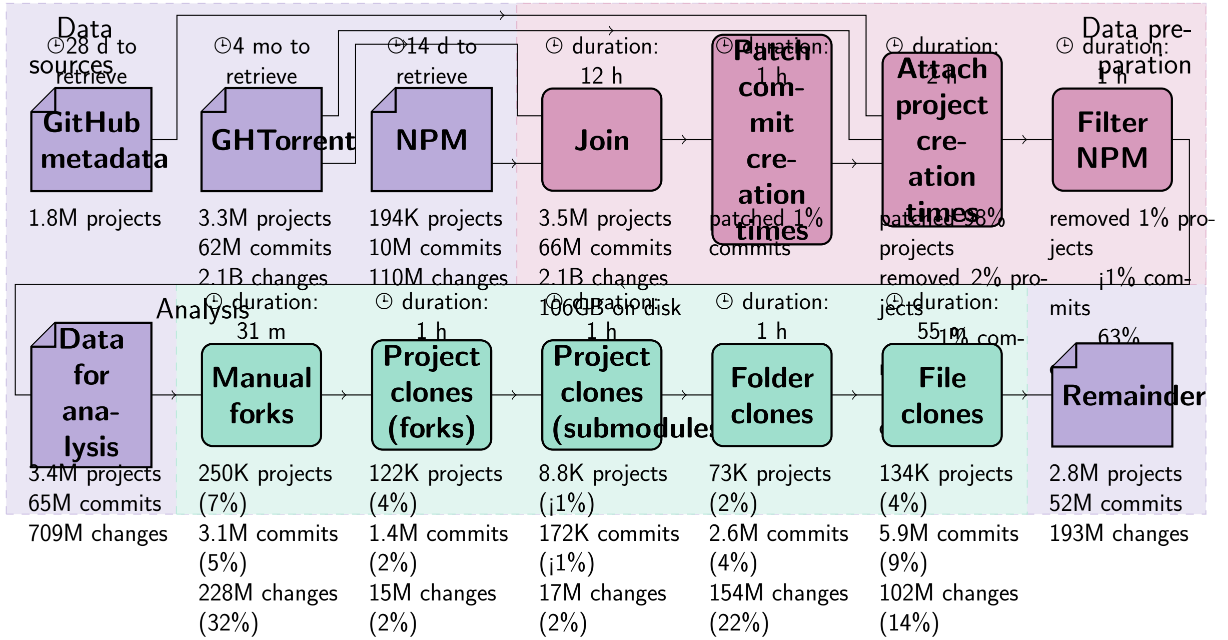

Figure 3.1: Data processing and analysis pipeline.

commits, and 193M changes. This data set consists of 83% of the cleaned up data set in terms of projects, 79% in terms of commits, and 27% in terms of changes.

The actual analysis took the least amount of time, less than 5 hours on the machine used for data preparation.

Results The detailed analysis confirmed the results we reported in [10]. Interestingly, we see that while cloning is indeed rampant, it is not evenly distributed across projects with only a few projects being responsible for very large volumes of cloned files. As already described by [10], most of the cloned files come from the inclusion of the NPM packages used by a project to its source code. Other reasons for cloning include, perhaps surprisingly re-pushing whole projects, including their history to different repositories, which we dubbed manual forking, as opposed to automatic forking facilitated by GitHub (such projects were already excluded from the dataset). Folder and single file clones are also reasonably populat at 22% and 14% of the remaining dataset.

Not only does the cloning affect only a minority of projects, it also affects only a minority of files. Figure 3.2 breaks all file versions into *unique* , which were observed only once, originals, and copies.

In total, the effects of cloning are signifficant: After all clones are removed, we ended up with 85% of the projects, 79% of commits, but only 9% of file versions.

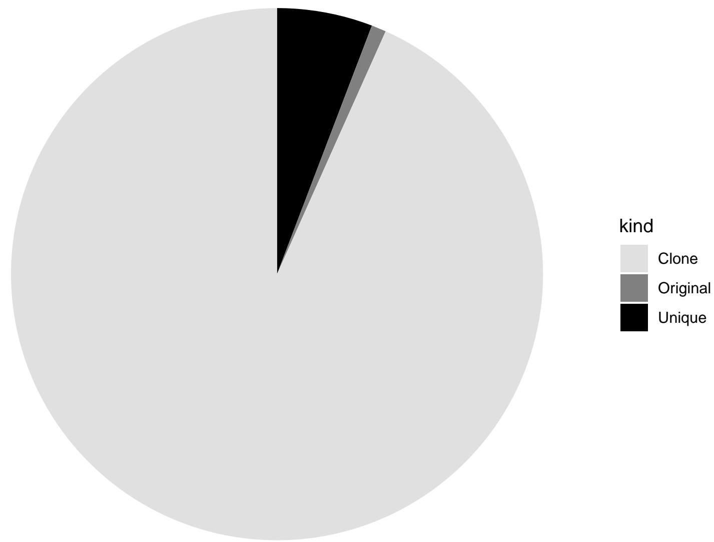

Figure 3.2: File versions broken into unique, original and copies.

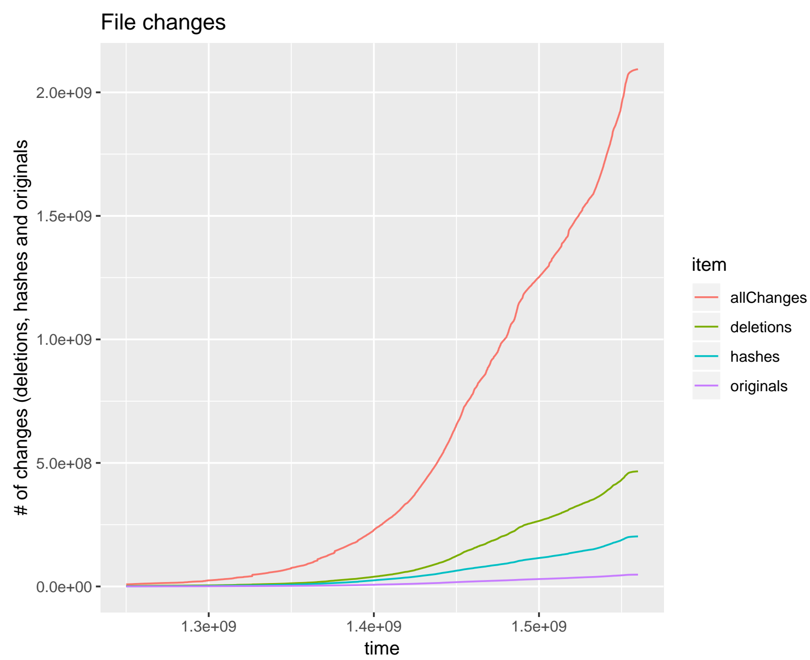

Figure 3.3: Changes in GH projects over time

3.2.2 Project Diversity

TODO bad name, do not know a better one, not sure it belongs in the thesis though, does it?

Talk about how the projects composition is pretty different too and how only a handful of the projects is actually interesting. This can be extended with the spider webs we created for the codedj2 but did not use?

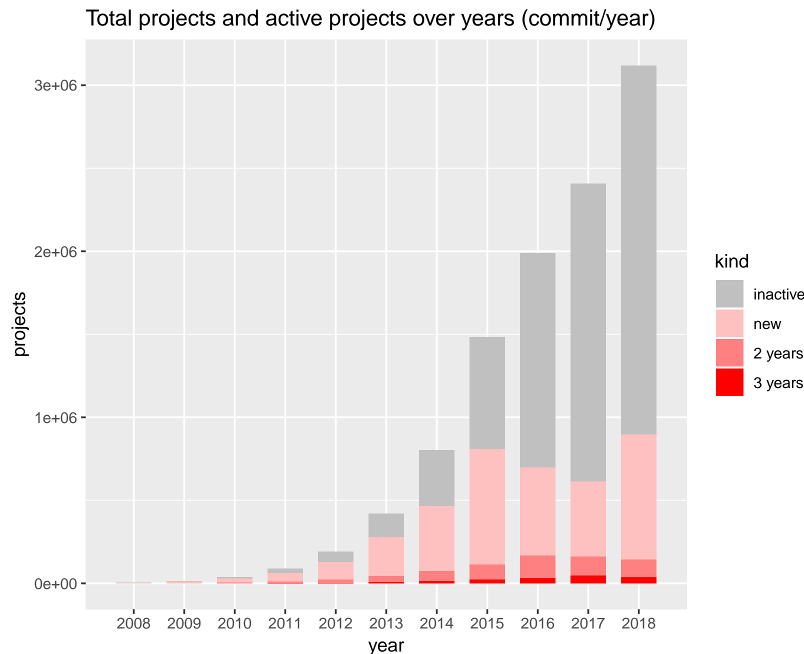

Figure 3.4: Lifespan of projects with at least one commit per year

3.3 CodeDJ

This section introduces CodeDJ and summarizes its main features, details of which can be found in [A.2]. The design of CodeDJ flows from four high-level principlest:

- Consistent, eventually: The sheer size and churn in data sources such as GitHub means that obtaining a snapshot of the whole data source is not practical. But, it is often the case that a slightly out-of-date view is sufficient for most investigations. We choose to refresh entire projects atomically at irregular intervals. Thus, any individual project is consistent, but for any group of projects, the lower bound on their refresh times is the last consistent time point (git histories can be destructively updated, allowing for post factum inconsistencies, we ignore these).

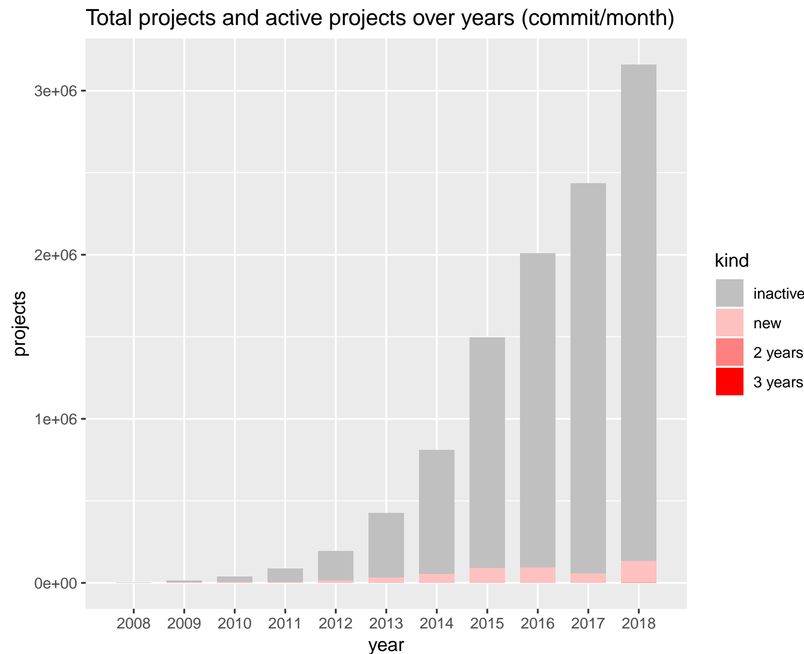

Figure 3.5: Lifespan of projects with at least one commit per month

- Code-centric, language agnostic: We aim to support queries on project metadata and file contents written in any programming language. To reduce space requirements, the only source artifacts we store is code, deduplication is used to remove redundancy, and metadata is trimmed where possible.

- Flexible query interface: Popular data science tools such as dplyr [19] or Spark [20] offer a mix operations inspired by database query languages extended with general purpose capabilities. Inspired by these, we propose an interface expressed in Rust as a library with operations for selecting, grouping, filtering and sampling data. The benefits of our approach over, say, SQL, is that queries are type-safe and benefit from the full generality of the Rust language.

- Reproducible by design: The importance of reproducibility cannot be overstated [5], consider [15] which recorded the names of the most starred projects seven years ago, without author names it is not possible uniquely to identify projects, and even with their full names, reconstructing a historical star count is not possible. CodeDJ is designed so it is possible to run any query with the information that the datastore had at an arbitrary point in the past. For this purpose the datastore is time-indexed, strictly append-only.

3.3.1 Querying

The querying method and its expressivity is an important aspect from user's perspective. Designing such interface is a ballancing act between user friendlines and expressivity as well as between complexity and generality of the queries: A simple query language as used by GitHub is trivial to express, but offers only very limited queries to be executed. Extension of the conjunctive queries to all attributes stored in the dataset helps, but cannot ever match dedicated query languages. Those can either be existing solutions, such as the SQL, or domain specific solutions. Using existing querying languages means that users will likely be already familiar with their concept. Furthermore they often exist within proven data storage solutions, such as databases that can be reused. They may however limit the expressivity or needlessly complicate the queries. As an example consider searching for all projects that use JavaScript as their main language and call the `eval` function. Furthermore, their existing storage solutions may not provide features required by CodeDJ dataset needs. On the other hand, domain specific languages such as Boa, if well designed, offer excellent query expresiveness and their backends can be tailored towards our specific needs. The price for their usage is the gamble (and extra work) placed upon their correct design of our side and the necessity to learn them for CodeDJ users.

As to what data should be queryable, the simple answer would be *everything*. However, one can only query what the system understands and so the more complex and detailed the queries, the more work is required from CodeDJ. Recall that Boa, the most complex project's search engine that allows searching within parsed source code files had to limit itself to a single language to provide the functionality. We believe that limiting CodeDJ to a single, or handful of, languages would be severely limiting to its users as exciting research happens in all kinds of programming languages and it is often the less popular languages where the task of finding relevant projects is the hardest. CodeDJ is thus intended to start as, and remain language agnostic. Querying file contents should be possible, but only a general pattern matching should be allowed.

Aside from searching and ordering, the query method used should support sampling. As the data volumes increase, it may not be feasible to analyze *all* projects fullfilling the query. At a minimum, a true random sampling, feature not available in any of the existing solutions, should be supported.

Decoupling Dataset Maintenance and Querying The most innovative idea of CodeDJ in terms of querying is however its complete decoupling of querying from the dataset maintenance. The actual querying API of CodeDJ is a simple linear scan and random access to all of the data items stored in the database with minimal overhead. This API will then be consumed by clients, that may leverage the CodeDJ infrastructure and guarantees while providing tailored search interfaces. CodeDJ already contains a reference search engine called Djanco, which was not developed by the author of this thesis and whose description can be found in the CodeDJ paper[A.2].

3.3.2 Parasite

Parasite is a dedicated, perpetually running application whose task is to synchronize its on-disk representation with GitHub. This task is complicated by these four constraints:

- Scalability: as the dataset is expected to eventually grow to hundreds of millions of projects, the storage and memory requirements must be kept compact and fast to access.
- Peaceful co-existence: Parasite must co-exist with GitHub peacefully, abiding by its terms of service.
- Robustness: As the size of the dataset grows, snapshots or dedicated backups will become impossible. So will working copies. The datset Parasite updates must at the same time support querying.
- Reproducibility: No data must ever be lost. Historical reproducibility must be baked in, i.e. the dataset must support being viewable at any given time in the past with no overhead.

Parasite's storage format is designed for minimal overhead, while maintaining fast linear and random access times. The information Parasite stores is broken down to multiple key-value pairs and those are stored in dedicated storage files. The files are append only, i.e. no information is ever lost. The storage files are optimized for efficient updates and forward linear scanning so that variable sized items are supported.

As an example, a project is broken into the following entities, each saved in its separate storage file:

- project information - consists of the project url, creator name, created time, etc.
- project heads - lists branches scanned for the project, their names and their head commits at the time the project was acquired. A project may be acquired multiple times, in which case a new visit simply add newer record of project heads
- commits - basic commit information (message, author, time, parents) and the changed files (paths and content hashes for all changes)
- snapshots - mapoping from a contents hash to actual file contents

In reality, more storage files are used. Paths and users are given their own storage files. All keys are replaced with integers and special storage files mapping the identifiers to the key values are added for commits and snapshots. Project metadata is split into multiple storage files and so on.

Indexes While the storage files contain all the information in the dataset, they are extremely inefficient for random access and deduplication necessary for their updating when keys are converted to identifiers. For fast random accesses, index files are used heavily. Any storage file can be accopanied with an index file that in its simplest form provides offsets to the storage file for any id at predefined positions. Unlike storage files, index files can also be modified and most importantly are not part of the dataset proper. Different clients of the same dataset may keep different index files that suit their needs.

Snapshots Parasite supports snapshotting via special snapshot files. Unlike traditional database snapshots, these do not contain the actual contents, but due to the append-only nature of storage files, a snapshot only needs to remember for each storage file its length. Snapshots can be created manually, or automatically and Parasite furthermore ensures that when a snapshot is taken, the datastore is in a valid state. Valid state means that all keys used in the dataset have their corresponding values stored as well such as for instance if a commit is in the database, its parents must be there too and so on. The simplest way to ensure validity is to only ever snapshot between project scanning, not in the middle. However, indexing very large projects, which can take hours to complete would then severely limit the snapshot granularity. Instead, when a project is scanned, it is scanned in a fashion that always keeps the dataset valid.

Whenever the dataset is accessed, a valid snapshot must be specified. Since the database is append only, this ensures that new projects can be added while the dataset is used with no data corruption. Furthermore, since all snapshots ever created become part of the dataset, the snapshot information also ensures historial reproducibility.

Finally, snapshots protect the dataset from failures. When the Parasite's scanner is restarted it verifies the latest snapshot agains the state of the dataset. If a discrepancy is detected, the dataset is rolled back to the last snapshot by clipping the excess file contents. Some data is lost, but this is unobservable from user's point of view as users can never access the data *after* the latest snapshot. Parasite makes sure that when orderly shut down a new snapshot is created when needed.

Schema updates As Parasite is expected to remain relevant for many years while preserving historical reproducibility, the dataset must deal with schema updates. Similar to the append-only data updates, schema is also append only, in the sense that a schema of existing storage file cannot be altered, but new storage files can be added at any time. With snapshoting, this ensures simple and robust mechanism as old clients accessing new data do not have to worry about new storage files and new clients accesing old data will see the new storage files as having length 0.

Initialization Acquiring the entirety of GitHub takes time. The Software Heritage has been around for years and still does not have all of GitHub. However, for many analyses, it is not necessary to have all of GitHub and a large enough portion would do. To this end, Parasite as presented in our paper used initial seed from GHTorrent to determine projects

belonging to a prioritized list of languages that were then downloaded. However, when analyzing the data, we have discovered that inconsistencies with GHTorrent are too large to be dismissed [A.1]. Current version of Parasite thus uses GitHub alone. We scan the public repositories endpoint to obtain the urls of all public repositories. We then randomly pick repositories for which we obtain metadata information, which gives us among other things the language of the project. Finally, a random subset of those projects belonging to the languages of interest is acquired. As both samples are increasing over time, eventually all of GitHub will be acquired, while at any point, the dataset remains an usable random subset.

Chapter **4**

Main Results

This chapter presents the 4 papers that together present the thesis contributions. The chapter opens with an overview of the research presented and timeline of the papers, followed by the papers themselves. Each paper is accompanied by a short clarification of the author's contributions.

4.1 Overview

We start with the foray into analysis of large corpora that poignantly illustrates the problem of bias in automated analysis of big code. We conducted an analysis of all non-forked public repositories available from GitHub in four major languages. However, as getting project urls from GitHub API is hard, we used GhTorrent for those first. Our aim was to look at the percentage of code duplicities (talk about different kinds and why they are bad). Report results.

With our knowledge, we wanted to see how much the huge duplication numbers can affect research results. For this we decided to conduct a replication of paper claiming interesting results with some non-trivial statistics. We have chosen the FSE paper to familiarize ourselves with these issues. During this we observed many problems with large code analysis (talk them all here). The single most alarming lesson was that reproducibility of historical papers is extremely hard. Reproducibility crisis.

Our next work thus concentrated on providing a service both for us and for other researchers that would help with the reproducibility issues of big code research. This revolved around creating a local copy of relevant Github information and providing a reproducible and scalable means of scrapping new information, retaining historical records and reproducible queries. CodeDJ. Using such a tool provides many benefits for researchers as it helps identifying and removing biases from their corpora. We demonstrated this on showing how the selection of projects to analyze can introduce variations to paper conclusions, for which we repurposed the previously done replication paper.

Noting that selecting by stars was not very representative of the whole GitHub corpora, for our next paper we updated CodeDJ to support more queries and datasets and

concluded an analysis of recent MSR papers and their selection methods, as well as informal reproduction of three papers using stars compared to a more robust random sample from an explicit selection supported by CodeDJ. (Stars are bad paper).

TODO should citations be included here, or at the very end, or both?

4.2 Paper 1 - DejaVu: A Map of Code Duplicates on GitHub

Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek.

In: Proc. ACM Program. Lang. 1, OOPSLA, Article 84 (October 2017), 28 pages. https://doi.org/10.1145/3133908

4.2.1 Author Contributions

The author of this PhD thesis was responsible for the following contributions in this paper:

The author was responsible for most of the JavaScript pipeline: data acquistion, adaptation of the tools used for Javascript and their improved scalability to support JavaScript sizes (some of which were later used for the other three languages as well), Almost all data visualization and graphs in the paper including the heatmaps and their analysis.

The author also implemented I also designed and implemented a framework in the R programming language that automatically generated all numbers used anywhere in the paper, which we used in subsequent papers as well.

4.3 Paper 2 - On the Impact of Programming Languages on Code Quality: A Reproduction Study

Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek.

In: ACM Trans. Program. Lang. Syst. 41, 4, Article 21 (October 2019), 24 pages. https://doi.org/10.1145/3340571

4.3.1 Author Contributions

I was responsible for the paper's main idea, analysis of the original paper, review and reproduction of the paper's artifact, data acquisition and reporting.

Due to the sensitivity of the topic, the paper's authors were presented in alphabetical order.

4.4 Paper 3 - CodeDJ: Reproducible Queries over Large-Scale Software Repositories

Petr Maj, Konrad Siek, Alexander Kovalenko, Jan Vitek.
In: 35th European Conference on Object-Oriented Programming (ECOOP 2021). Article No. 7; pp. 7:1–7:24

4.4.1 Author Contributions

I was responsible for the paper's main idea, the overall design of the system and its components. the design and implementation of parasite, the reproduction analysis (without the graphs) and I helped a bit with the query language interface design.

4.5 Paper 4 - The Fault in Our Stars: How to Design Reproducible Large-scale Code Analysis Experiments

Petr Maj, Stefanie Muroya, Konrad Siek, Jan Vitek

4.5.1 Author Contributions

I was responsible for the paper's main idea, acquisition and curation of all datasets, dataset analysis, stars analysis, most extensions of the Djanco Query engine, one replication experiment and supervision of Stefanie.

4.6 Discussion

4.7 Summary

Chapter 5

Conclusions

TODO I'll write this when I have the rest mostly ok-ish...

5.1 Summary

5.2 Contributions of the Dissertation Thesis

TODO again the template says this should happen three times, so here it is:

1. Analysis of cloning and associated biases in large collections of software repositories (paper 1)

2. Analysis of reproducibility issues and statistical interpretation of large corpora. Proposal of better methodology for reproducibility (paper 2)

3. Design and implementation of a tool for large scale download, archival and querying of software repositories to aid reproducble project selection and analysis (pape 3)

4. Analysis of the selection bias introduced by the most frequentlty used project selection by popularity convenience sampling on recent papers. Analysis of the obtained and missed projects and development of methodology for reproducible and validable project selection and associated tooling (paper 4)

5.3 Future Work

An obvious improvement would be in the scalability of the presented tools. While CodeDJ queries run fast enough for a typical research problem, and authorized researchers can be given direct access to the database and servers for more ad-hoc query processing, such approach will not scale for use outside of academia.

Currently, only GitHub is supported as data source. CodeDJ has already been created with modularity in mind and supports addition of extra sources, although none has been implemented yet. Any git backed VCS will be almost trivial and others can be adapted with relative ease as none of git objects survive into the datastore itself. Primary candidates are archives (such as Software Heritage), other source code hosting providers (Sourceforge, BitBucket, etc.), or source package managers (NPM, crates.io, etc.). Those additional sources will have varying metadata available, which the query engine would have to account for.

Selecting by stars has already been discredited enough. And while the thesis provides a general methodology for its replacement, this is by no means complete. More detailed directions and in-depth analysis of various methods should be attempted.

TODO something else?

Synthesized attributes Schema updates in Parasite are expected to bemore frequent than as necessitated by addion of new GitHub features.

Bibliography

[1]

[2] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality: A reproduction study. *ACM Trans. Program. Lang. Syst.*, 41(4), oct 2019.

[3] T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere. Orion: A software project search engine with integrated diverse software artifacts. In *International Conference on Engineering of Complex Computer Systems*, 2013.

[4] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES 2017: 14th International Conference on Digital Preservation*, Kyoto, Japan, 2017.

[5] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *International Conference on Software Engineering (ICSE)*, 2013.

[6] Davide Falessi, Wyatt Smith, and Alexander Serebrenik. Stress: A semi-automated, fully replicable approach for project selection. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017.

[7] Jesus M. Gonzalez-Barahona, Gregorio Robles, and Santiago Dueñas. Collecting data about FLOSS development: The FLOSSMetrics experience. In *International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS)*, 2010.

[8] Georgios Gousios and Diomidis Spinellis. GHTorrent: GitHub's data from a firehose. In Michael W. Godfrey and Jim Whitehead, editors, *Working Conference on Mining Software Repositories (MSR)*, 2012.

[9] Github LLC. The 2021 state of the octoverse, 2021.

[10] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1:1 – 28, 2017.

[11] Petr Maj, Konrad Siek, Alexander Kovalenko, and Jan Vitek. CodeDJ: Reproducible Queries over Large-Scale Software Repositories. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:24, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[12] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100, sep 2017.

[13] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 155–165, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

[15] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *International Conference on Software Engineering (ICSE)*, 2016.

Reviewed Publications of the Author Relevant to the Thesis

[A.1] P Maj, S Muroya, K Siek, J Vitek *The Fault in Our Stars: How to Design Reproducible Large-scale Code Analysis Experiments* to be accepted somewhere.

[A.2] P Maj, K Siek, A Kovalenko, J Vitek *CodeDJ: Reproducible Queries over Large-Scale Software Repositories* 35th European Conference on Object-Oriented Programming (ECOOP 2021) 2021.

[A.3] ED Berger, C Hollenbeck, P Maj, O Vitek, J Vitek *On the impact of programming languages on code quality: a reproduction study* ACM Transactions on Programming Languages and Systems (TOPLAS), vol 41, issue 4, pages 1-24 2019.

[A.4] C. Lopez and P. Maj and P. Martins and V. Saini and D. Yang and J. Zitny and H. Sajnani and J. Vitek *DéjàVu: A Map of Code Duplicates on GitHub*. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) 2017.

Remaining Publications of the Author Relevant to the Thesis

[A.5] ED Berger, P Maj, O Vitek, J Vitek *SE/CACM Rebuttal [2]: Correcting A Large-Scale Study of Programming Languages and Code Quality in GitHub* arXiv preprint arXiv:1911.11894 2019,

[A.6] P. Maj and C. Hollenbeck and S. Hussain and J. Vitek *Analyzing Duplication in JavaScript.* BenchWork 2018.

[A.7] P. Maj and F. Gauthier and C. Hollenbeck and S. Hussain and J. Vitek and C. Cifuentes *Building a node.js Benchmark: Initial Steps.* BenchWork 2018.

[A.8] P Maj *Analyzing Large Code Repositories.* Ph.D. Minimum Thesis, Faculty of Information Technology, Prague, Czech Republic, 2018.

Remaining Publications of the Author

[A.9] J Sliacky, P Maj *Lambdulus: teaching lambda calculus practically* Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E, pages 57-65 2019.

[A.10] T. Kalibera and P. Maj and F. Morandat and J. Vitek *A Fast Abstract Syntax Tree Interpreter for R.* Conference on Virtual Execution Environments (VEE), 2014.

[A.11] P Maj, T Kalibera, J Vitek *TestR: R language test driven specification* The R User Conference, useR! July 10-12 2013 University of Castilla-La Mancha, Albacete, Spain, vol 10, issue 30, pages 149 2013.

[A.12] T Kalibera, J Hagelberg, P Maj, F Pizlo, B Titzer, J Vitek *A family of real-time Java benchmarks* Concurrency and Computation: Practice and Experience, vol 23, issue 14, pages 1679-1700 2011.

[A.13] F Pizlo, L Ziarek, E Blanton, P Maj, J Vitek *High-level programming of embedded hard real-time devices* Proceedings of the 5th European conference on Computer Systems, pages 69-82 2010.

[A.14] F Pizlo, L Ziarek, P Maj, AL Hosking, E Blanton and J Vitek *Schism: fragmentation-tolerant real-time garbage collection* ACM Sigplan Notices, vol 45, issue 6, pages 146-159 2010.