Otto-von-Guericke University Magdeburg

Faculty of Computer Science

Master Thesis

Query-Analysis on Git-Attributes in Relational and Graph-DB

Author:

Aditya

March 4, 2022

Advisors:

Prof. Dr. Gunter Saake

Department of Technical and Business Information Systems

Dr. David Broneske

German Centre for Higher Education Research and Science Studies

Dr. Jacob Krüger

Ruhr-Universität Bochum

Aditya:
Query-Analysis on Git-Attributes in Relational and Graph-DB
Master Thesis, Otto-von-Guericke University Magdeburg, 2022.

Abstract

GitHub being the largest collaborative hosting platform for software engineering projects provide researchers an opportunity to analyze the publicly available data to answer interesting research questions, for example, 1) Code evolving rate, 2) Topics occurring most in desktop web-app development, 3) Interactions distinguishing between developers joining/remaining outside project teams. All these questions try to achieve the same goal, i.e., to find answers in data. Data management is the crucial topic here, because the decision on how the big data from GitHub is managed has effects on the planned evaluation and the analysis on the data. For example, the data structure lying underneath the data has effects on performance of queries executed on the dataset. Many initiatives, such as GHTorrent/GHArchive, have been introduced in the past to support researchers in the data management — However, the successful ones of them provide data packed in the relational data model and none of them provides data management solution as a graph database on which researchers can execute graph queries efficiently. This is an interesting context, because of the following reasons, 1) GitHub stores the Version Control System (VCS) history in graph data structure, 2) GitHub attribute relations can be structured as a forest, 3) Graph algorithms have their strong part in computer science for finding answers for problems that fit well in a graph data structure.

In this thesis, it has been confirmed that the *big* data researchers collect from GitHub mostly lacks proper data management, i.e. data schema and DBMS. There are exceptions where the relational data model emerges. Nevertheless, the use of the other data models, such as the graph data model, rarely comes in the focus.

In order to open new research perspectives and define a base ground for future research questions, 1) We evaluate papers from the Mining Software Repositories (MSR) conference to define GitHub attributes relevant to researchers, 2) We define the data models and the DBMS that are used by researchers in their analysis, 3) We estimate if it is possible to transform the researchers' analysis in SQL queries to define the evaluation from the view point of the relational model.

Contents

List of Figures

List of Tables

1. Introduction

GitHub is based upon the git version control system and provides a collaborative code hosting service. Researchers worldwide use publicly available software engineering data on GitHub for conducting their empirical studies. They put up research questions, such as, 1) code evolving rate in GitHub [Manes and Baysal [2021]], 2) topics occurring most in desktop web-app development [Scoccia et al. [2021]], 3) interactions distinguishing between developers joining/remaining outside project teams [Middleton et al. [2018]]. All the research questions in the GitHub context try to achieve the same goal, i.e., to find answers in data.

To achieve this goal, the *big* data is often collected from the GitHub API that does not conform to any particular data schema, but instead the API supplies results in JSON. Hence, data curation, filtration, and management becomes a task for the researchers with which they have to engage themselves - without getting answers to their research questions from the effort spent on this part.

Data management is the crucial topic here, because the decision on how the big data from GitHub is managed has effects on the planned evaluation and the analysis on the data itself. For example, the data structure lying underneath the data has effects on performance of queries executed on the dataset. Many initiatives have been introduced in the past years to support the researchers for the data management part (see the related work section for more information on this). However, the successful and famous ones of them provide the GitHub data packed in the relational data model and none of them provides data management solution as a graph database on which researchers can execute graph queries efficiently. This is an interesting context, because of the following reasons, 1) GitHub stores the Version Control System (VCS) history in graph data structure, 2) GitHub attribute relations can be structured as a forest as depicted in Figure 1.1, 3) graph algorithms have their strong part in computer science for finding answers for problems that fit well in a graph data structure.

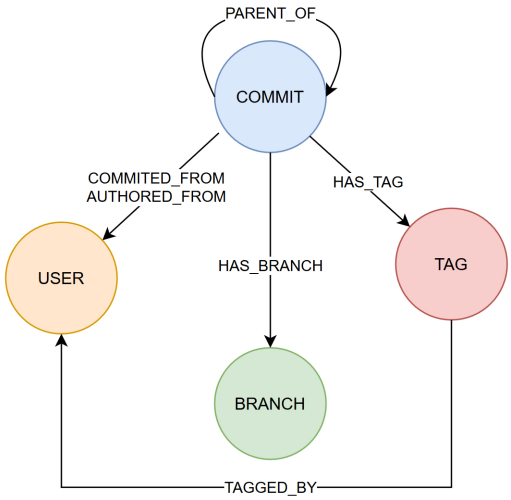

Figure 1.1: GitHub attributes and their relations expressed as a forest

Not losing the balance focusing *only* on the graph data structure and instead generalizing the issue: The potential topic of research arises as, if data collected from GitHub needs *another* data model than the relational data model. In other words, if there is any use case of using any other data structure for storing GitHub data in a database. Would researchers' evaluation and analysis benefit from such a data structure?

To this end, 1) we need to define the scenario in which researchers study GitHub data, 2) we need to define the data models and DBMS that are *already* in use, 3) we need to estimate the possibility of transiting data management for software engineering research analysis to a DBMS. The relevance of these investigations is described in the following.

- The first investigation filters out the relevant GitHub attributes. This is important for deciding on the ideal data model, as depending on the type of attributes, combinations of attributes can be well-structured in one model but not in the other.
- The second investigation filters out the data models and DBMS already in use. This is relevant, because before suggesting or finding an ideal solution to a problem, it is wise to give a check on the solutions already in use.
- The third investigation specifies the possibility of using a DBMS. This is relevant, because data management is well associated with a database instance in a DBMS. Since data management is the main concern in this thesis, this investigation is unavoidable.

Contribution of this Thesis: The aforementioned three investigations are the three pillars underlying the contribution of this thesis and give birth to our three research questions. Hence, our contribution can be summarized as answers to the following research questions.

- **RQ1** *Relevant GitHub Attributes*: Which data attributes from GitHub (for example: commits, timestamps) are relevant for researchers in the Software Engineering community?
- **RQ2** *Available Data Models and DBMS*: Which data models or database management system (DBMS) do researchers use for storing their data?
- **RQ3** *SQL transformation*: Is it possible to transform the researchers' evaluation on GitHub attributes in form of SQL queries?

The thesis is structured as follows. Chapter 3 describes the literature review methodology. Chapter 4 presents the evaluation of the papers that are selected using the methodology described in Chapter 3. In Chapter 5 we discuss and present the findings for the evaluated papers. Chapter 6 describes the related work. Chapter 7 presents the conclusion.

2. Literature Review Methodology

For the selection of the studied papers, the Mining Software Repositories (MSR) conference was chosen [MSR]. The 2008 and 2011 proceedings of the MSR conference had an overall 43% acceptance rate [ACT], which served as a quality criterion for selecting this conference. The papers studied were all accepted during the course of 2018 till 2021.

Describing the MSR conference community in brief, the conference welcomes new models, tools, or techniques which support modelling of software data, its analysis, or mining procedures. The goal being to solve problems in software engineering (SE) through making common practices in SE better, as the research in the area could potentially support decision-making and predictions in software development projects, validation of techniques and ideas already existing in SE, improvement for reuse and designing of software, software systems maintenance, and software behaviors for users along with its evolution and from the focal point of runtime. This is achieved through analysis of data and hence uncovering of information and patterns which result in improved comprehension of software development processes (including human and social aspects) in SE. Whereby origins of data - among other sources - being version controlling systems like GitHub, or question-answer websites like Stack Overflow. Few examples of type of data include 1) CI logs, 2) release information, 3) programming language features, 4) software licenses and copyrights, and 5) code review data.

Defining the MSR conference further, the major data source in MSR papers is according to Flint et al. [2021] summarized as the GitHub platform. This concludes the major data type to be of version control systems. During the course of studying MSR papers, the experience was made that often no explicit mentioning of using a DBMS or a data model occurs, this lies also fit with the trend of not explicitly mentioning any data filtration techniques for data curation in the majority of the MSR papers as concluded by Flint et al. [2021].

Other than the first criterion of the papers being accepted, the second condition for them was that they exist in the collection of *technical papers* which is different as

of the collection of *mining challenge* or *data showcase*. While the technical papers collection comprises the papers, which present a solution to a stated problem for prompting action [TEC], the collection representing the Mining Challenge is focused on a common dataset, which is annually pre-defined by the MSR organization [MIN]. The concept of the common dataset might skew the flexibility range for the researchers during planning their research questions and research techniques. Therefore, for the sake of a wider scope of research methodologies, papers from the Mining Challenge collection were not considered. On the other hand, the collection of Data Showcase papers represent papers, which bring in new datasets for other researchers for their future interest [DAT]. The negative aspect for considering the Data Showcase papers in this thesis would be that these papers do not collect data or do evaluation on data from the perspective of finding answers to research questions. Instead they are agnostic to any particular research question and their data collection deem to only serve future interests often not clear at the moment of publishing of the Data Showcase papers.

The third criterion for the paper selection was a temporal criterion, i.e. the papers must not be older than three years at the time of writing this thesis. This methodology gives us papers in the time range of the year 2018-2021. This temporal aspect was important to define because GitHub is to be considered as an ever-changing web-service platform. Hence, we advocate that papers published much before in time might not be as relevant to the current GitHub features and attributes, such as GitHub Actions, as the papers published in the last three years.

The fourth most and the final criterion for the paper selection was regarding the title of the respected papers. The requirement was made that a paper's title must include the term *git*. This implies for us that the authors of the respected paper give git or GitHub high priority for their research either for their data collection or answering their research questions.

The following list summarizes the selection criteria; 1) papers accepted to MSR, 2) papers belong to technical papers collection, 3) papers are not older than three years, 4) paper's title include term *git*.

3. Evaluation

3.1 Papers from Mining Software Repositories Conference 2021

In this section, we evaluate the papers published in the year 2021 at MSR conference.

3.1.1 TNM: A Tool for Mining of Socio-Technical Data from Git Repositories

Text retrieval for analysis of sociotechnical data from GitHub is a challenging task. To this end, Sviridov et al. [2021] introduce in their paper *TNM: A Tool for Mining of Socio-Technical Data from Git Repositories* a tool for data mining.

Relevant GitHub Attributes: with respect to **RQ1** the paper discusses their tools' functionalities which uses data attributes from GitHub for their working methodologies. There are 5 functions named as FilesOwnershipMiner, CommitInfluenceGraphMiner, AssignmentMatrixMiner and FileDependencyMatrixMiner, WorkTimeMiner, and ChangedFilesMiner. Whereby, for example, FilesOwnershipMiner as a functionality tries to calculate the knowledge of a developer, hence it requires access to attributes such as *author's name* and *changed files* from corresponding author. Table 3.1 represents the summarized accessed attributes from the paper. The forthcoming list gives a full description of such functionalities which access GitHub attributes.

1. FilesOwnershipMiner has the Degree of Knowledge (DOK) algorithm from Carlson [2015] as its core, the algorithm calculates the knowledge of a developer or a set of specified developers about a predefined section in code. The generation of a knowledge scoring between particular developers and file pairs in the structure of a nested-map and the line-level code authorship data is the primary mining function. This functionality of the tool requires access to: *author's name* and *changed file(s) name* from that author.

2. CommitInfluence-GraphMiner as illustrated by Suzuki et al. [2017] tries to apply PageRank algorithm to commits in GitHub repositories to specify commits' influence. The modified application of the algorithm replaces Web pages with commits and hyperlinks with causal relationships between commits. Especially bug-creating commits and bug-fixing commits stand in the focus of the applied version of the algorithm, because such commits are considered to be the most influencing commits, having strong causal relationships among them. As a step of the algorithm, bug-fixing oriented commits need to be identified. To this end, the term "fix" is searched in commit messages. Afterwards, Git blame feature is used to find earlier commits, which changed same code lines during their submission, as the ones changed in the bug-fixing commit. Hence, the relationships between the bug-creating and bug-fixing commits is identified by the algorithm. The format of the result is a map of lists, where keys represent commit hash of bug-fixing commits and values represent commit hash of bug-creating commits. This functionality of the tool would require access to: *commit messages*, *changed code lines (aka code diff)*, and *commit hash*.
3. AssignmentMatrixMiner and FileDependencyMatrixMiner functionality create data for socio-technical congruence analysis. A coordination gap exists, if there is a need for coordination among developers for writing or reviewing code lines. The aforementioned socio-technical congruence analysis helps to reduce coordination gaps by either decreasing the causes for coordination or by supporting coordination. AssignmentMatrixMiner generates a modification count for every developer and file pair as a nested map. This concludes to the data that gets used at a later time to calculate socio-technical resemblance. FileDependencyMatrixMiner searches for the files those were altered in a commit and generates the number of times these files have been modified for every file pair. The utilization of this data leads to the building of edges in one of the socio-technical software network parts, which is established on the Conway's law from Conway [1968]. This functionality of the tool would require access to: *author's name*, *changed file(s) name*, and *commit hash*
4. WorkTimeMiner gathers how, in the duration of week, commits get distributed in a time series. The usage of such data can lead to a number of profits. An example of a use case could be to improve scheduling work through intersections search in the time series among non-similar developers. This functionality of the tool would require access to: *commit hash* and *time of commit*.
5. ChangedFilesMiner can be used for mining altered files for every developer. Furthermore, it can be used for identifying files that are edited from more than a single developer. This functionality of the tool would require access to: *author's name* and *changed file(s) name*.

Available Data Models and DBMS: in correspondence to **RQ2**, no DBMS is used for data management. Nevertheless, the data models used for storage are 1) JSON for maps from Mapper interface, 2) nested map for developer and file pairs, 3) map of lists for commit hash as keys and lines changed as values. There are other data storing procedures present, whereby the data structure for those are unclear or not defined in the scope of the paper.

SQL transformation: regarding **RQ3**, the researchers' evaluation on GitHub attributes can be transformed in SQL queries, however the requisites existing that the dataset first needs to be stored in the relational model and tables structure belonging to data schema are to be defined.

Table 3.1: Summarized accessed attributes in the paper from Sviridov et al. [2021]

Author Name
Changed Files Name
Commit Messages
Changed code lines (aka code diff)
Commit Hash
Commit timestamp

3.1.2 Studying the Change Histories of Stack Overflow and GitHub Snippets

The paper *Studying the Change Histories of Stack Overflow and GitHub Snippets* from Manes and Baysal [2021] also served answers in part to our research questions. The paper discusses the code's evolving patterns for comparison of code changes and trends. They try to analyze the change pattern among snippets from Stack Overflow and GitHub community. For this purpose they work with two datasets namely SOTorrent and GHTorrent.

Relevant GitHub Attributes: regarding **RQ1**, the paper uses data attributes from GitHub for the creation of their code-change history dataset named as GHCodeSnippetHistory. For example, for identification of changes based on commits, commit hash is used; and for checking compliance with copyright, clone URL request is sent to GitHub API. Table 3.2 represents the summarized accessed attributes from the paper.

Available Data Models and DBMS: with respect to **RQ2**, the relational model is used for storing data as the GHCodeSnippetHistory dataset. Furthermore, GHTorrent as a publicly available dataset is used, which also relies on the relational model. Even though, no explicit mentioning of a DBMS occurs in the paper, it is assumed that MySQL DBMS is used for GHTorrent dataset, because the GHTorrent project provides automated SQL scripts for easily setting up their datasets in MySQL DBMS.

SQL transformation: regarding **RQ3**, the transformation of the researchers' evaluation on GitHub attributes in SQL queries is possible, because the datasets are available in the relational model. Even though the schema of the relational model for the GHCodeSnippetHistory dataset is not well-defined, there exists a proper documentation for the GHTorrent dataset. For the formulated SQL queries (Table 3.2) the GHCodeSnippetHistory dataset was modified and no use of the GHTorrent dataset was necessary, as the required attributes of GHTorrent dataset were present as subset in the GHCodeSnippetHistory dataset. With respect to the pattern of the the queries, the queries represent only a single pattern, i.e. no-joins (Table 3.3).

Table 3.2: Summarized accessed attributes in the paper from Manes and Baysal [2021] along with their SQL queries and patterns

commit hash (dataset: GHCodeSnippetHistory) select Hash from CommitDB; Pattern: no-join
parent commits (dataset: GHCodeSnippetHistory) select ParentCommit from CommitDB; Pattern: no-join
project size (dataset: GHCodeSnippetHistory [modified]) select ProjectSize from CompletedPostReferenceGHTable; Pattern: no-join
number of files (dataset: GHCodeSnippetHistory [modified]) select filesNumber from CompletedPostReferenceGHTable; Pattern: no-join
project clone URL data not available for analysis (but could be composed using automation scripts or requesting GitHub's API)
user name (dataset: GHCodeSnippetHistory [modified]) select user from CompletedPostReferenceGHTable; Pattern: no-join
project name (dataset: GHCodeSnippetHistory [modified]) select repoName from CompletedPostReferenceGHTable; Pattern: no-join

Table 3.3 gives an overview of *all* Query-Patterns in the paper from Manes and Baysal [2021]

Table 3.3: overview of *all* Query-Patterns in the paper from Manes and Baysal [2021]

2-way-join: 0 times
distinct-search: 0 times
no-join: 6 times
string-search: 0 times
3-way-join: 0 times
grouping: 0 times
complex-query: 0 times
self-join: 0 times
using-NULL: 0 times

union-multiple-queries: 0 times
multiple-where-conditions: 0 times
counting: 0 times

3.1.3 How Do Software Developers Use GitHub Actions to Automate Their Workflows?

Evaluation of new GitHub features, like GitHub Actions, which allows repository maintainers to automate their workflow up to an extent, have achieved little focus through researchers. In the paper from Kinsman et al. [2021] *How Do Software Developers Use GitHub Actions to Automate Their Workflows?*, they evaluate how developers use GitHub Actions and how its adoption alters the activity indicators.

Relevant GitHub Attributes: corresponding to **RQ1**, the list of accessed basic attributes is presented in Table 3.4 and computed ones in Table 3.5. The following describes the listed attributes in the aforementioned tables.

The authors access attributes classifiable in two types, 1) pull request related properties, 2) general project properties. For example, for the identification of active repositories, *number of pull requests opened* are considered. Whereby *programming language* property of repositories is used for grouping repositories and presenting the results for usage of GitHub Actions through group classification. Additionally, variables like *project name* and *programming language* are used to project variability among different projects and languages using statistical modelling techniques.

Available Data Models and DBMS: with respect to **RQ2**, there is no evidence of using a DBMS in the paper. Nevertheless, the use of the relational data model occurs because the authors derive their data from the RepoReapers dataset (containing only a single table), made available through Munaiah et al. [2017]. Additionally, API calls to GitHub have been made to identify the repositories using GitHub actions, although the data model used for storing the results of such calls has not been mentioned. The authors performed calculations, for example, aggregation on many variables such as number of pull requests, for answering their research question regarding the impact of GitHub Actions. Nevertheless, data models used for storing the variables or the results of calculations on those variables is not mentioned in the paper. While the results of such calculations have been used for statistical modeling using the Regression Discontinuity Design, it is unclear in which data model the statistical model has been provided the input.

SQL transformation: regarding **RQ3**, the transformation in form of SQL queries is possible, because the RepoReapers dataset contains data in CSV format, which is easily transferable to a relational table in any RDBMS.

Table 3.4: Summarized accessed basic attributes in the paper from Kinsman et al. [2021]

Merged/non-merged pull requests
Project name
Programming language
Total number of commits
Number of pull requests opened Time-to-merge/time-to-close pull requests
Time since the first pull request
Total number of pull request authors

Table 3.5: Summarized accessed computed attributes in the paper from Kinsman et al. [2021]

Issues containing keywords "github action" or "github actions" + with existing comment and posted after GitHub Actions feature release
Comments on merged/non-merged pull requests
Commits of merged/non-merged pull requests

3.1.4 Escaping the Time Pit: Pitfalls and Guidelines for Using Time-Based Git Data

In the paper *Escaping the Time Pit: Pitfalls and Guidelines for Using Time-Based Git Data* from Flint et al. [2021] the authors claim that methodologies of a number of MSR-papers depend on time-based data. The authors demonstrate potential problems with such data. Afterwards, the quantification of occurrences of such data follows.

Relevant GitHub Attributes: regarding **RQ1**, the list of accessed basic attributes is presented in Table 3.6 and computed ones in Table 3.7. The following describes the listed attributes in the aforementioned tables.

Attributes such as *commit timestamps* play a major part in queries executed by the authors on their dataset. As for instance, if a timestamp is too old, it might be suspicious for being wrong data. Other attributes such as *commit parents* are also

used for evaluations, for example, if a commit has a parent and the same commit is older in time than its parent, then it turns out to be wrong data again. For expressing word-clouds containing words typically used in bad commits, i.e. commits with wrong time based data, the attributes such as *commit messages* are used. For sorting bad commits by projects containing those commits and their authors, the paper describes a ranking system for which attributes such as *commit author* and *project name* are used.

Available Data Models and DBMS: with respect to **RQ2**, no evidence of using a DBMS is found in the paper. The Boa Infrastructure [BOA] is used for quantification and identification of wrong commit timestamps. For example, a query in Boa outputs for wrong timestamps their corresponding commit ID and project URL. Nevertheless, this infrastructure does provide data outputs agnostic to a data model. Figure 3.1 depicts an example output. Furthermore, the authors use their own scripts to filter and analyze data - instead of executing SQL on a dataset packed in a DBMS. An example of such a script command from grep tool for counting occurrences of specified text is *grep 'text' file.txt | wc -l* [Flint et al. [2021]]. Additionally, the GitHub API and the Software Heritage [SWH] is used for the data collection in JSON format.

```
counts[] = java, 50136, 0
counts[] = c++, 40375, 0
counts[] = php, 32378, 0
counts[] = c, 30308, 0
counts[] = python, 15117, 0
counts[] = c#, 15034, 0
counts[] = javascript, 12569, 0
counts[] = perl, 9740, 0
counts[] = unix shell, 4314, 0
counts[] = delphi/kylix, 3811, 0
```

Figure 3.1: output model of Boa-Query [BOA]

SQL transformation: regarding **RQ3**, it is not easily possible to transform the researcher's evaluation on GitHub attributes in form of SQL queries, because there exists no particular data model which could be translated into the relational model. Nevertheless, the data can still be modeled as relational but would lack at first a sensible schema, as none is used for it in the paper.

Table 3.6: Summarized accessed basic attributes in the paper from Flint et al. [2021]

Parent commits
Commit hash
Commit timestamp
Project URL
Commit messages
Author name
Project name

Table 3.7: Summarized accessed computed attributes in the paper from Flint et al. [2021]

Total number of commits (on basis author)

3.1.5 Challenges in Developing Desktop Web Apps: A Study of Stack Overflow and GitHub

In the paper *Challenges in Developing Desktop Web Apps: A Study of Stack Overflow and GitHub* from Scoccia et al. [2021], the authors conclude challenges faced by developers while integrating desktop-frameworks for web-apps. To this end, the authors use topic-modelling for mining Stack Overflow datasets in the context of web-apps desktop development. For confirming their results, they match emerging issues on Stack Overflow with issue reports present in the corresponding GitHub repositories.

Relevant GitHub Attributes: with respect to **RQ1**, the list of accessed basic attributes is presented in Table 3.8. The following describes the listed attributes in the aforementioned table.

Commit information, i.e. *commit hash* and its *timestamp* are used to identify active repositories. For example, the requirements to filter their dataset were, 1) at least 10 commits in repository, and 2) time span of 8 weeks among first and last commit in repository. To confirm that repositories in their dataset represent real-world projects, they also collected attributes such as *stars* and *watchers* per project. Additionally, many attributes in the context of issue reports were collected, such as its *title*, *status*, *edit dates*, etc. The set of attributes collected for issue reports is large, because the evaluation on the GitHub part mainly relates to the context of issue reports.

Available Data Models and DBMS: with regard to **RQ2**, no explicit mentioning of using a DBMS occurs in the paper. Hence, it is assumed that the data is stored and processed locally using ad-hoc programming or shell scripts. In the data collection section of the paper, the authors mention using such ad-hoc scripts for gathering application-lists data, whereby during the collection they also apply some filters such as the data must contain a link to GitHub repository. Nevertheless, it remains unclear in which data model the collected data was shaped into. In total, data for 114,710 issue reports is collected, although the data model for the storage of the strongly intra-related attributes in relation to the issue reports (i.e., their title, text, author, labels etc.) is also not mentioned. In the pre-processing phase the authors perform stopwords removal, for example, ”at”, ”as”, ”is”, for reducing noise in the data. Additionally, lemmatization and stemming is performed on the data. Nevertheless, it remains unclear if any particular data model is used during the aforementioned noise removal procedures. Furthermore, the Latent Dirichlet Allocation algorithm and tests such as the Anderson-Darling test, Friedman test, and Nemenyi test are applied to their data. However, no explicit mentioning of using a data model occurs for feeding input to the aforementioned algorithms or tests.

SQL transformation: regarding **RQ3**, it is not easily possible to transform the researcher's evaluation on GitHub attributes in form of SQL queries, because there

exists no particular data model which could be translated into the relational model. Nevertheless, the data can still be modeled as relational but would lack at first a sensible schema, as none is used for it in the paper.

Table 3.8: Summarized accessed basic attributes in the paper from Scoccia et al. [2021]

Commit hash
Project name
Commit timestamp
Issue reports
Issue reports title
Text on issue discussion page
Issue reports author
Issue reports labels
Issue reports current status
Issue reports creation / submission timestamp
Issue reports last edit / closing timestamp
Stars for projects
Watchers per project

3.2 Papers from Mining Software Repositories Conference 2020

In this section, we evaluate the papers published in the year 2020 at MSR conference.

3.2.1 A Study of Potential Code Borrowing and License Violations in Java Projects on GitHub

In the paper *A Study of Potential Code Borrowing and License Violations in Java Projects on GitHub* from Golubev et al. [2020], the authors check repositories containing Java code for code plagiarism. Their study is well in time, because of the increasing production of open source tools and the production code being publicly accessible on GitHub. Java is in focus for their study, because 1) licence violations in cloning of Java code can have legal actions as consequence, and 2) Java is a popular programming language.

Relevant GitHub Attributes: with respect to **RQ1**, the list of accessed basic attributes is presented in Table 3.9. The following describes the listed attributes in the aforementioned table.

The information about *project forks* is used, for example, to exempt forks from the code plagiarism checking. This step was important because code borrowing within forks of a repository does not constitute as plagiarism. Moreover, *code language* as an attribute of a repository was used to filter projects containing Java code, because plagiarism checking of Java code was in the focus of the study. Furthermore, the code plagiarism procedure used in the paper requires the attribute *commit timestamps*

as part of its functionality. The attribute *project name* was assessed to reduce redundancy in the dataset, because the authors used the Public Git Archive (PGA) [Markovtsev and Long [2018]] dataset for their study that contains a number of projects with different names but lead to the same repository. The attribute *project licence* was also assessed, because the plagiarism checking procedure requires knowing a project's licence.

Available Data Models and DBMS: regarding **RQ2**, the following data models are concluded from the study of the paper, 1) file-tree data model, and 2) labelled graph data model. The file-tree data model is used for data storage, because the major source of the data collection was PGA. PGA allows downloading repositories as zip package, whereby the structure of files is of file-tree model. Under common database terms, the file-tree model is to be defined as the hierarchical datastructure, where each parent node is the upper directory linking (through edges) the child directories. In this model either the files themself can be the nodes (not recommended because of security and performance reasons) or the nodes contain only the pointer to the files, whereby the original files reside on the disk. The graph data model is used by algorithm processing code blocks for checking licence violations. In the procedure, edges represent clone pairs and nodes represent code blocks. As mentioned in the paper, several algorithms for the aforementioned processing were considered, whereby all of them were designed to use the graph data model. The timestamp of the last modification in a code block was used to turn undirected edges between code blocks to directed ones. For example, code block 11.01.2021 → code block 14.01.2021. For each code block the licence associated with it (or with the file/project) was assigned to the graph model and the code block pairs were labelled as permitted, prohibited or permissive.
There might be other data models in use for the management of repository files. Nevertheless, no evidence for them is found in the paper. For example, after filtration procedures, the final dataset comprised of 233,378 repositories. It remains unclear if any data model, for example, document oriented database model, was used for managing so many repository files. In case no document oriented database model was in use, it is assumed that all these repository files were organised in a simple file-tree model. With regard to the previous statement a document oriented model is to be defined as modeling documents in JSON-like structure [AWS] that differentiates it from the file-tree model. Additionally, the full VCS history of these repositories was collected and in total 1.55 TB of disk space was consumed. Using a DBMS in such a case could have been proved efficient. Nevertheless, no explicit mentioning of using a DBMS occurs.

SQL transformation: corresponding to **RQ3**, it is not easily possible to transform the researchers' evaluation on GitHub attributes in form of SQL queries, because it does not make sense to translate file-tree data model to the relational model directly. Nevertheless, the documents' content that are structured in the file-tree model can be transformed in such a way that it can be packed into the relational model. The labelled graph data model used for analysis in the paper can be translated into the relational model, given that the corresponding algorithm need to be adapted to the relational model. The algorithm might need to be composed completely from scratch, because changing the underlying data structure for an algorithm leads to the same

effort as changing the algorithm itself. Nevertheless, the detailed description of exact working steps of the aforementioned algorithm is missing in the paper.

Table 3.9: Summarized accessed basic attributes in the paper from Golubev et al. [2020]

Code language
Commit timestamp
Project name
Project forks
Project license

3.2.2 The State of the ML-universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub

In the paper *The State of the ML-universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub* from Gonzalez et al. [2020], the authors study 4,524 applications and 700 AI/ML tools as a *community* to highlight its unique trends and development patterns in comparison to the non-AI/ML repositories.

Relevant GitHub Attributes: with respect to **RQ1**, the list of accessed basic attributes is presented in Table 3.10 and computed ones in Table 3.11. The following describes the listed attributes in the aforementioned tables.

The attribute *project topics* was used to filter repositories that are associated with AI/ML. 439 topic labels were used for this purpose, such as, *tensorflow*, *adversarial machine learning*, and *natural language processing*. The attribute *project description* was used to categorize AL/ML repositories into subclasses, i.e. tools or applications. This means that if the description of a project contains keywords such as library, tool, framework, toolkit, etc. then the repository is classified as a tool. In case parsing the description of a project was not able to classify a project, the corresponding *project GitHub page* was manually reviewed to do manual classification. The attributes *stars* and *commit timestamp* were used to select non-AL/ML repositories. As an example, the last commit timestamp must be within 2019 and must have more than or equal to 5 stars. The aforementioned filtration was necessary, because the set of non-AL/ML repositories is quite large. To confirm that the collected repositories represent active software projects and not, as for instance, home-work assignments, attributes such as *forks*, *project size*, *project name*, and once again *project description* and *stars* were accessed. One of the filtration criterion using the aforementioned attributes were that the size must be greater than 0 KB and the project name and description were manually reviewed to confirm that the content represents a software project. Some analyses classified the data further using the attribute *project owner type*, because an owner of a project can be a user or an organization.

Additional attributes were assessed to measure the number of collaborative interactions between users. The aforementioned measurements were for collaborations among users presented in Table 3.11.

Available Data Models and DBMS: regarding **RQ2**, the use of GitHub API and GHTorrent dataset implies the usage of JSON and the relational data model respectively. For example, the GitHub API was used to create a list of repository labels, hence it is assumed that this list was stored in JSON format on disk. Other than the models mentioned in the previous statement, no evidence of using any particular data model is present for most of the procedures used in the paper. Each AI/ML repository in the dataset was categorized as a tool or application, nevertheless it remains unclear how the categorization followed in the text of files. If categorization would have been done in the relational model, an additional column would have been defined. In case of the graph model, property of nodes would have to be defined. Nevertheless, it is assumed that no data model was used for the aforementioned purpose. To identify repositories representing a tool, the repositories in the dataset were matched with a popular tool-repositories list available on the internet. However, this list is in a free text form and hence does not constitute being in a particular data model. It is concluded that the matching procedure was not performed in any DBMS, instead programming scripts or shell commands were used for this purpose. The authors also mention parsing the description text of each repository in the dataset for search terms such as tool, framework, etc. Additionally, data was parsed to count the number of times users have particular roles. However, it is unclear how the parsing was exactly performed. Probably, command-line tools such as grep might have been used for the purpose. The repositories in the dataset were also filtered according to their size, number of stars, number of forks, commit timestamp, etc. Nevertheless, it is not mentioned if execution of any DBMS query aided in the filtration. Hence, it is assumed that again programming scripts might have been used here. With regard to the GHTorrent dataset, the authors explicitly mention creating an automated script to parse parts of the dataset. Concluding from this statement, it is assumed that GHTorrent dataset was not only used as database packed in a DBMS, but also the data in the file format was parsed using programming scripts for specialized use cases, such as creating a record of complex collaborative interactions. As a result of the research, many facts were presented about the data studied. For example, the oldest AI/ML repository dates back to the year 2009. Furthermore, many procedures calculated medians for using it in results. Nevertheless, it remains unclear if any particular data model was used to perform calculations efficiently. Additionally, tests such as Kruskal-Wallis test and Dunn's test were applied on the data, whereby no explicit usage of any data model is mentioned to feed these tests.

SQL transformation: corresponding to **RQ3**, it is in part possible to transform the researchers' evaluation on GitHub attributes in form of SQL queries, because GHTorrent dataset with the relational data model was used in the paper. Table 3.10 and 3.11 represent the formulated SQL queries and their patterns. Table 3.12 gives an overview of the query patterns.

Table 3.10: Summarized accessed basic attributes in the paper from Gonzalez et al. [2020] along with their SQL queries and patterns

Project name
select name from projects;
Pattern: no-join

Project description
select description from projects;
Pattern: no-join

Project GitHub page
select url from projects;
Pattern: no-join

Project size
select name, bytes from projects, project_languages
where projects.id = project_languages.project_id;
Pattern: 2-way-join

Stars
Data not available for analysis

Forks
select a.name, count(a.forked_from) from projects a, projects b
where a.forked_from = b.id group by a.name;
Pattern: self-join / grouping / counting

Commit timestamp
select id, created_at from commits;
Pattern: no-join

Contributors
select distinct login from users;
Pattern: no-join / distinct-search

Project topics
Data not available for analysis

Project owner type
select type from users, projects
where projects.owner_id = users.id;
Pattern: 2-way-join

Table 3.11: Summarized accessed computed attributes in the paper from Gonzalez et al. [2020] along with their SQL queries and patterns

Commit author and committer
(under aspect: Contribution)
(committer and author of a commit are considered as collaborators)
select author_id , committer_id from commits;

Pattern: no-join

Issues commentator
(under aspect: Discussion)
(issue's commentators are considered as collaborators
if none of them is the reporter of the issue)
select user_id from issue_comments, issues
where issue_comments.issue_id = issues.id
and user_id <> reporter_id;
Pattern: 2-way-join / multiple-where-conditions

Commit commentator
(under aspect: Discussion)
(commit's commentators are considered as collaborators
if none of them is the author of the commit)
select user_id from commit_comments, commits
where commit_comments.commit_id = commits.id
and user_id <> author_id;
Pattern: 2-way-join / multiple-where-conditions

Pull requests commentator
(under aspect: Discussion)
(pull request's commentators are considered as collaborators
if none of them is the opener of the pull request)
select user_id from pull_request_comments,
pull_requests, pull_request_history where
pull_requests.id = pull_request_comments.user_id
and pull_request_history.id = pull_requests.id
and pull_request_history.action <> "opened";
Pattern: 3-way-join / multiple-where-conditions / string-search

Commit commentator
(under aspect: Review)
(commit's commentators including the author of the commit
are considered as collaborators)
select user_id from commit_comments, commits
where commit_comments.commit_id = commits.id;
Pattern: 2-way-join

Issue collaborator
(under aspect: Process)
(issue's reporter and user initiating
an event for maintenance are considered as collaborators)
select distinct login from users, issue_events, issues
where issues.reporter_id = users.id and
issue_events.actor_id = users.id and
issue_events.issue_id = issues.id;

Pattern: 5-way-join / distinct-search / multiple-where-conditions

Pull request collaborator
(under aspect: Process)
(pull request's opener and user initiating
an event for maintenance are considered as collaborators)
select pull_request_id, action, users.id
from pull_request_history, users where
users.id = pull_request_history.actor_id;
Pattern: 2-way-join

Pull request collaborator
(under aspect: Maintenance)
(if users start a maintenance event and
none of them is the pull request's opener,
they are considered as collaborators)
select pull_request_id, action, users.id from
pull_request_history, users where users.id =
pull_request_history.actor_id and action <> "opened";
Pattern: 2-way-join / multiple-where-conditions

Table 3.12: overview of *all* Query-Patterns in the paper from Gonzalez et al. [2020]

2-way-join: 7 times
distinct-search: 1 times
no-join: 6 times
string-search: 1 times
3-way-join: 1 times
grouping: 1 times
complex-query: 0 times
self-join: 1 times
using-NULL: 0 times
union-multiple-queries: 0 times
multiple-where-conditions: 4 times
counting: 1 times

3.2.3 Need for Tweet: How Open Source Developers Talk About Their GitHub Work on Twitter

Fang et al. [2020] in their paper *Need for Tweet: How Open Source Developers Talk About Their GitHub Work on Twitter* combine their analysis on Twitter and GitHub

to uncover patters of GitHub users tweeting. For example, tweeting patterns of repository owners that differentiate them from the other repository users or pattern of users who follow a repository, tweet about the repository, but do not contribute any other way to the project (i.e. pull requests, issue opening or commits).

Relevant GitHub Attributes: regarding **RQ1**, the list of accessed basic attributes is presented in Table 3.13 and computed ones in Table 3.14. The following describes the listed attributes in the aforementioned tables.

The attribute *user profile page URL* was assessed for mining Twitter URL explicitly given by a user on his profile page. *User email* as another attribute aided the identification of Twitter accounts possessing same the email address and hence belonging to the corresponding GitHub user. Study results concluded that project owners tweet mostly for advertising. For analysis behind this result, the attribute *project owner* was assessed. Furthermore, attributes presented in Table 3.14 were accessed for analysis resulting that tweets from the aforementioned users relate to work discussion and to answering questions.

Available Data Models and DBMS: in relation to **RQ2**, GHTorrent dataset concludes the use of the relational data model and calls made to the GitHub API suggest storing resulting data as JSON data model. In this non-exhaustive small paper (5 pages), no evidence of using another data model occurs.

SQL transformation: corresponding to **RQ3**, it is in part possible to transform the researchers' evaluation on GitHub attributes in form of SQL queries, because GHTorrent dataset with the relational data model was used in the paper.

Table 3.13: Summarized accessed basic attributes in the paper from Fang et al. [2020]

User profile page URL
User email
Project owner

Table 3.14: Summarized accessed computed attributes in the paper from Fang et al. [2020]

User closing issue (:*collaborator*)
User closing pull request (:*collaborator*)
User opening pull request (:*contributor*)
User committing (:*contributor*)
User opening issue (:*contributor*)
User follows project owner or *collaborator* or *contributor*

3.3 Papers from Mining Software Repositories Conference 2019

In this section, we evaluate the papers published in the year 2019 at MSR conference.

3.3.1 Predicting Good Configurations for GitHub and Stack Overflow Topic Models

In the paper *Predicting Good Configurations for GitHub and Stack Overflow Topic Models* from Treude and Wagner [2019], the authors study configurations for the Latent Dirichlet Allocation (LDA) algorithm to be able to predict ideal configurations for future unseen datasets. Describing the LDA, the algorithm groups texts to suggest topics. The authors use datasets of GitHub and Stack Overflow in their study. Although, our interest does not go beyond GitHub platform (i.e. to Stack Overflow), one of the study results defined that LDA needs different configurations for GitHub and for Stack Overflow.

Relevant GitHub Attributes: with regard to **RQ1**, the list of accessed basic attributes is presented in Table 3.15. The following describes the listed attributes in the aforementioned table.

The project's ReadMe file played a vital role in the study, because it is expected from the content of this file that it describes the project. Hence, the description text should aid topic modelling process of LDA. To this end, *project's ReadMe files* were accessed as an attribute. Moreover, this file was not collected for all projects available on GitHub. Instead, the requisite was made that the project must be containing code in one of the programming language chosen by the authors. In this context, the attribute *project's programming languages* was accessed.

Available Data Models and DBMS: regarding **RQ2**, no evidence of using any DBMS occurs. Other than the JSON data model for data returned from the API for data collection, no usage of any data model occurs. For the creation of the GitHub corpora, 5,000 documents for each of the 8 selected programming languages were stored. Nevertheless, it remains unclear if any document oriented DBMS was used for the management of these 40,000 documents. The authors explicitly mention using a script for randomly picking a project during their data collection and to carry operations further on the chosen project and its ReadMe file. For example, if the ReadMe file had no non-ASCII characters and had more than 100 characters, they included it in their dataset for further evaluation. The usage of a non-DBMS script, in the aforementioned case, suggests (in general, i.e. not specifically for LDA application) using no DBMS queries and hence no SQL. The dataset was preprocessed before evaluation, for example, line breaks, HTML tags, comments, etc. were removed. It remains unclear, if this preprocessing was performed in any DBMS or data was moulded in any particular data model beforehand for efficient processing.

SQL transformation: corresponding to **RQ3**, it is not easily possible to transform the researchers' evaluation of GitHub attributes in form of SQL queries, because no particular data model is used for storing the data. Even though there would exist a lacking schema, the content of the documents in the dataset can still be transformed in the relational model.

Table 3.15: Summarized accessed basic attributes in the paper from Treude and Wagner [2019]

Project's ReadMe File
Project's Programming Languages

3.3.2 git2net - Mining Time-Stamped Co-Editing Networks from Large git Repositories

Gote et al. [2019] in their paper *git2net - Mining Time-Stamped Co-Editing Networks from Large git Repositories* introduce their tool for constructing co-editing networks by evaluating co-authorship within files. Their approach differs from the majority of the related work, because their tool focuses on the code line level (instead of file level) and hence produces fine-grained networks. To be exact, their tool produces the following types of networks 1) directed acyclic graph (DAG) for sequences of edits, 2) bipartite graph with set of developers and files, and 3) co-editing graphs.

Relevant GitHub Attributes: with regard to **RQ1**, the list of accessed basic attributes is presented in Table 3.16. The following describes the listed attributes in the aforementioned table.

The attribute *commit hash* was used to identify commits as an entity. For each such an entity, further attributes were collected. For example, information about the author of that commit, i.e. date on which author committed (*author date*), timezone to which that date belongs (*author timezone*), *author email*, *author name*, name of the branch to which committing (*branch name*). The attributes relating to an author in the previous statement were analogously collected for a committer, i.e. *committer date*, *committer email*, *committer name*, *committer timezone*. Furthermore, boolean type attributes were collected, informing *if committing* in main branch of a project or *if merging* is the case. Attributes of integer type were, for example, number of *modifications*, length of commit message (*commit message length*). Additionally, *parents* of a commit and name of the project (*project name*) were also gathered for each commit. All this aforementioned collection of attributes were necessary to create networks of co-editing.

Available Data Models and DBMS: regarding **RQ2**, the authors explicit mention that they use SQLite database for storing the result of tool's procedures. This concludes to the usage of the relational data model and the sqlite DBMS. No evidence of using another data model occurs for the procedures of the tool. Nevertheless, the tool also outputs resulting graph networks in CSV format for their visualization through external network analysis tools.

SQL transformation: corresponding to **RQ3**, even though the usage of sqlite database and the relational data model occurs, it is not easily possible to transform the researchers' *original* evaluation of GitHub attributes in form of SQL queries. The original evaluation on the attributes used the pydriller package [Spadini et al. [2018]] for data collection and authors' written code for analysis and evaluation on the collected data. The procedures store their *resulting* output in sqlite database in structured relational tables. Hence, an artificial mirror evaluation can be performed

in form of SQL queries on the resulting dataset. The original evaluation can still be translated in SQL queries, given that the results of the pydriller part in the procedures are first stored in the relational model.

Table 3.16: Summarized accessed basic attributes in the paper from Gote et al. [2019]

commit hash
author date
author email
author name
author timezone
branch name
committer date
committer email
committer name
committer timezone
if in main branch
if merging
modifications
commit message length
parents
project name

3.3.3 Identifying Experts in Software Libraries and Frameworks among GitHub Users

In the paper *Identifying Experts in Software Libraries and Frameworks among GitHub Users* Montandon et al. [2019] use Machine Learning (ML) along with other techniques, such as clustering, and mining LinkedIn profiles, to identify experts among GitHub users for libraries and frameworks. They limit their demonstration to 3 JavaScript libraries. Additionally, as ground truth, they also conduct a survey among the users in their dataset, i.e. they asked them to rate their expertise in the aforementioned libraries. Through their study, they confirmed that ML was not a good technique for their evaluation. Nevertheless, their clustering results and survey conclude that it is indeed possible to analyse expertise of users in selected libraries and recommend them based on such an evaluation.

Relevant GitHub Attributes: with regard to **RQ1**, the list of accessed basic attributes is presented in Table 3.17 and computed ones in Table 3.18. The following describes the listed attributes in the aforementioned tables.

The authors explicitly mention using only the attributes which are directly accessible from the GitHub API, whereby all attributes represent different kind of changes on the data. As for instance, both attributes in Table 3.17 represent quantitive changes on the data. On the other hand, attributes in Table 3.18 represent temporal changes and variable changes. Under variable changes is to understand changes that are performed not in a single project but in a broader range of projects. The authors defend their choice of aforementioned attributes as they perceive analysing

the expertise of a user as a well mixed combination of quantitive, temporal and variable changes contributed from such a user.

Available Data Models and DBMS: corresponding to **RQ2**, the use of the JSON data model emerges clearly, because the authors explicitly mention using the GitHub API for their data collection. Nevertheless, it remains unclear, if JSON-formatted data was moulded in another data model before filtration. As examples for such filtration procedures, aliases of users were removed from the users list (list comprising around 10,000 users), and projects filtration based on their dependency on a specific library. Furthermore, two sets were mapped to each other, i.e. set of users and set of GitHub accounts. This procedure could be represented as a variant of the bipartite graph. Hence, the graph data model could have been used here for efficient processing. Nevertheless, no usage of such a data model (or the term bipartite graph) occurs for the aforementioned use case. Additionally, during the final processing of data, missing values for certain GitHub attributes were replaced (based on specific criteria) with 0. For such a replacement of values, SQL could have been used if data were in the relational model. However, it is not specified how the aforementioned procedure was executed. For the correlation measurement and the removal of correlated GitHub attributes, *cor* function from R's *stats* package and *findCorrelation* function from R's *caret* package were used. Again, no specific data model is explicitly mentioned for feeding these functions.

SQL transformation: regarding RQ3, it is not easily possible to transform the researchers' evaluation of GitHub attributes in form of SQL queries, because JSON, instead of the relational data model is used for storing the data. However, all the collected attributes are fetched from the GitHub API. Hence, even though no use of GHTorrent or any other publicly available dataset occurs in the paper, still GHTorrent with the relational data model can be used for mirroring the evaluation (to limited extent) in form of SQL queries.

Table 3.17: Summarized accessed basic attributes in the paper from Montandon et al. [2019]

Number of commits (: *quantity of changes*)
Number of added import lines (: *quantity of changes*)

Table 3.18: Summarized accessed computed attributes in the paper from Montandon et al. [2019]

Number of commits altering at least one file (: *quantity of changes*)
Number of commits adding import line (: *quantity of changes*)
Number of lines added or removed in all commits (as code churn variant) (: *quantity of changes*)
Number of lines added or removed in all files (as code churn variant) (: *quantity of changes*)
Number of days since first commit adding import line (: *frequency of changes*)
Number of days since last commit adding import line (: *frequency of changes*)
Number of days *between* first and last commit adding import line (: *frequency of changes*)
Average days gap between commits changing files (: *frequency of changes*)
Average days gap between commits adding import line (: *frequency of changes*)
Number of projects in which user contributed at least once (: *variability of changes*)
Number of projects in which user added import line (: *variability of changes*)

3.3.4 Striking Gold in Software Repositories? An Econometric Study of Cryptocurrencies on GitHub

Trockman et al. [2019] in their paper *Striking Gold in Software Repositories? An Econometric Study of Cryptocurrencies on GitHub* hypothesize that activity trend in cryptocurrency software development correlates with the price of the cryptocurrency. As base of the hypothesis, they consider software quality as one of the influencing factors of the cryptocurrency price. Nevertheless, their study's results discover no such correlation.

Relevant GitHub Attributes: with regard to **RQ1**, the list of accessed basic attributes is presented in Table 3.19. The following describes the listed attributes in the aforementioned table.

To understand the mindset behind collecting the attributes listed in the aforementioned table, we first need to go through the first research question authors put up, i.e. if activity, popularity, and quality assurance is correlated with cryptocurrency price? This leads to accessing the GitHub attributes in three categories, i.e. activity, popularity, and quality assurance. The attributes are marked according to these categories in Table 3.19. In this short paper (5 pages) no other GitHub attributes played a role.

Available Data Models and DBMS: regarding **RQ2**, the GitHub API was used for the data collection on the GitHub side (with the exception of repository badges and continuous integration). This confirms using JSON format for the data storage. On the cryptocurrency-market side, data such as price and market capitalization information was collected from CoinMarketCAP API [CMC]. The CoinMarketCAP's API declares explicitly: all endpoints return data in JSON format. Hence, JSON gets used again as a data model. Afterwards, the collected data was moulded in the relational data model. Furthermore, the authors publicly make available their relational dataset.

SQL transformation: corresponding to RQ3, it is possible to transform the researchers' evaluation of GitHub attributes in form of SQL queries, because the authors moulded the collected data from the GitHub API in the relational data model.

Table 3.19: Summarized accessed basic attributes in the paper from Trockman et al. [2019]

Number of daily commits (: *activity*)
Contributors (: *activity*)
Lines of code added (: *activity*)
Lines of code removed (: *activity*)
Number of unique active developers (: *activity*)
Stars (: *popularity*)
Forks (: *popularity*)
Watchers (: *popularity*)
Repository badges (: *quality assurance*)
Continuous Integration (CI) (: *quality assurance*)

3.4 Papers from Mining Software Repositories Conference 2018

In this section, we evaluate the papers published in the year 2018 at MSR conference.

3.4.1 Large-Scale Analysis of the Co-Commit Patterns of the Active Developers in GitHub's Top Repositories

In the paper *Large-Scale Analysis of the Co-Commit Patterns of the Active Developers in GitHub's Top Repositories* from Cohen and Consens [2018], the authors try to discover patterns of co-commits among active developers and developers that are not so active. Under the term active is to understand a high frequency of commits. It is claimed by the authors that few developers in a repository are responsible for the majority of the commits. The authors also study how such co-commit patterns evolve over time. Furthermore, the co-committing networks of GitHub are compared with those of other platforms such as Wikipedia or scientific research portals.

Relevant GitHub Attributes: with regard to **RQ1**, the list of accessed basic attributes is presented in Table 3.20 and computed ones in Table 3.21. The following describes the listed attributes in the aforementioned tables.

The accessed attribute *stars* was necessary, because authors wanted to select repositories for their study based on their popularity. The attribute *commit timestamp* was accessed, because the formation of co-committing networks require time based data for directing one node to another node. These two attributes we consider as the basic attributes. Now coming to the description of computed attributes, the attribute *number of repositories per user*, *users participating in multiple language repositories*, and *number of languages per user* were assessed with respect to one of their research questions that was engaged with the discovery of co-commit patterns under the perspective of different programming languages. This research question brought along the necessity of observing the number of repositories per user, because often different repositories consist of a variety of different programming languages. The attribute *commits made by users for each language* was accessed to define how active a developer is, whereby the categorization followed through programming languages. The activity behaviour of developers was also analysed without the perspective of programming languages, hence the attribute number of commits per user was accessed. The attribute *number of commits per repository* was accessed because the activity threshold had to be defined as in relation to the average number of commits in a repository. For filtration purposes, for example, for repositories that have at least 2 developers, the attribute *number of committer per repository* was accessed.

Available Data Models and DBMS: regarding **RQ2**, the GitHub data was collected using the GitHub API. Furthermore, the authors explicitly mention mining commits from the commit logs, whereby commit logs themselves were also collected using the GitHub API. Hence, the use of the JSON data model is clear. The data was stored in a tuple format, for example, *(language, repository, developer, date, commits)*. This aforementioned storage was realized through CSV files, whereby the name of the programming language was overtaken as the file name and further attributes were the content of that file. In the analysis part of the study, the graph data model is used, whereby the network analysis tool NetworKit [Staudt et al. [2015]] was applied. The usage of the graph data model was important, because the analysis algorithms in NetworKit demanded a graph data structure. The graph procedures used were: 1) Connected Components, 2) Degree Distribution, 3) Network Centralization, 4)

Community Structure. Under *Threats To Validity* section, the authors mention using the Google's BigQuery GitHub data and GHTorrent dataset to validate their original collected data. Since both Google's BigQuery GitHub data and GHTorrent dataset are available in the relational data model, at last the use of the relational data model emerged.

SQL transformation: corresponding to RQ3, the collection of GitHub attributes can be easily transferred to the relational model, because the authors make their dataset publicly accessible in CSV format. However, it is not easily possible to transform the researchers' evaluation of GitHub attributes in form of SQL queries, because for the evaluation the graph data model in context of the NetworKit tool was used.

Table 3.20: Summarized accessed basic attributes in the paper from Cohen and Consens [2018]

Commit timestamp
Stars

Table 3.21: Summarized accessed computed attributes in the paper from Cohen and Consens [2018]

Number of repositories per user
Number of languages per user
Users participating in multiple language repositories
Commits made by users for each language
Number of commits per user
Number of commits per repository
Number of committer per repository

3.4.2 Which Contributions Predict Whether Developers Are Accepted Into GitHub Teams

In the paper from Middleton et al. [2018] *Which Contributions Predict Whether Developers Are Accepted Into GitHub Teams*, the authors try to discover patterns which differentiate developers who end up being a team member in open-source software projects and those who remain outsiders. To this end, they compare the forms and frequency of contributions from developers set.

Relevant GitHub Attributes: with regard to **RQ1**, the list of accessed basic attributes is presented in Table 3.22 and computed ones in Table 3.23. The following describes a subset of the listed attributes in the aforementioned tables.

To define if a developer is an insider or an outsider, the authors make use of the attribute *if user has write access*. In simple words, this translates to, if the user can commit. The attribute *contributors* is used for defining anyone, irrespective of the user being an insider or an outsider. This can be abstracted as all users who do any action related to the project, for example, commenting, watching, forking — all are to be considered as contributors. Under this aspect, the attributes *number of projects user watches*, *number of projects user forks*, etc. were accessed. Since the authors wanted to identify patterns of developers in relation to pull requests, they made sure that all users in their dataset have initiated at least one pull request. Hence, the attribute *if user initiated pull request* was accessed. The attribute *project team roles* was accessed, because this attribute can define exactly if a developer is a team member. Nevertheless, GitHub changed the feature related to this attribute during the research period of the authors. This led them to include further GitHub attributes in their study, such as *pull request discussions*, to define properly who is a team member and who is not. Similarly, the attributes *if user merging / closing pull request*, and *if user committing without pull request* were accessed, because only team members can perform these operations.

Available Data Models and DBMS: regarding **RQ2**, the GHTorrent dataset was used, hence the usage of the relational data model is clear. Along with the relational data model, the use of the MySQL DMBS explicitly emerges, as authors downloaded the MySQL dumps of the GHTorrent dataset. However, for data that was not available in the GHTorrent dataset, the authors collected the respective webpages from GitHub — it remains unclear, if these webpages themselves were stored on the disk in a particular data model.

SQL transformation: corresponding to **RQ3**, it is possible to transform the researchers' evaluation of GitHub attributes in form of SQL queries, because in major part GHTorrent is used. Table 3.22 and 3.23 represent these queries.

Table 3.22: Summarized accessed basic attributes in the paper from Middleton et al. [2018] along with their SQL queries and patterns

contributors
select distinct login from users;
Pattern: no-join / distinct-search

project team roles Data not available for analysis
stars Data not available for analysis

Table 3.23: Summarized accessed computed attributes in the paper from Middleton et al. [2018] along with their SQL queries and patterns

if user has write access (aka committer) select distinct login from users, commits where users.id = commits.committer_id; Pattern: 2-way-join / distinct-search
if user initiated pull request select distinct login from users, pull_request_history where users.id = pull_request_history.actor_id and action = "opened"; Pattern: distinct-search / 2-way-join / string-search
pull request discussions select distinct login from users, pull_request_comments where users.id = pull_request_comments.user_id; Pattern: distinct-search / 2-way-join
if user merging / closing pull request select distinct login from users, pull_request_history where users.id = pull_request_history.actor_id and action = "merged" or "closed"; Pattern: distinct-search / 2-way-join / string-search
if user committing without pull request select distinct login from users, pull_request_history, commits where users.id <> pull_request_history.actor_id and users.id = commits.committer_id; Pattern: distinct-search / 3-way-join
number of projects user watches select count(repo_id), login from watchers, users where watchers.user_id = users.id group by user_id; Pattern: grouping / 2-way-join
number of projects user forks with c as (select count(b.id) as forkedCount, a.owner_id as ownerID from projects a, projects b where a.forked_from = b.id group by a.owner_id) select forkedCount, login from c, users where users.id = c.ownerID;

Pattern: complex-query / grouping / self-join / 2-way-join

number of issues user opens in projects
with c as (select count(issue_id) as issuesCount, reporter_id as reportingUser from issues group by reporter_id) select issuesCount, login from c, users where users.id=c.reportingUser;
Pattern: complex-query / grouping / 2-way-join

number of comments user adds in issue discussions
select count(comment_id), login from issue_comments, users where users.id = issue_comments.user_id group by login;
Pattern: grouping / 2-way-join

number of pull requests user submits
select count(pull_request_id), login from pull_request_history, users where action = "opened" and pull_request_history.actor_id = users.id group by users.login
Pattern: grouping / 2-way-join / string-search

number of commit comments by user
select count(comment_id), login from commit_comments, users where users.id = commit_comments.user_id group by login;
Pattern: grouping / 2-way-join

comments to pull requests created by user
select count(comment_id), login from pull_request_comments, pull_request_history, users where action="opened" and actor_id = pull_request_comments.user_id and actor_id = users.id group by login;
Pattern: grouping / string-search / 3-way-join

comments to pull requests by user created by others
select count(comment_id), login from pull_request_comments, pull_request_history, users where action <> "opened" and actor_id = pull_request_comments.user_id and actor_id = users.id group by login;
Pattern: grouping / string-search / 3-way-join

tenure (time between user interaction / dataset snapshot)
Data not available for analysis

number of commits user made to own projects
select count(commits.id), login from commits, projects, users where commits.committer_id = users.id and users.id = projects.owner_id group by login;
Pattern: grouping / 3-way-join

number of non-forked projects user created
select count(projects.id), login from projects, users
where forked_from is NULL and owner_id
= users.id group by login;
Pattern: grouping / using-NULL / 2-way-join

number of users watched OR forked project from specific user
select count(distinct a.owner_id), b.owner_id
from projects a, projects b where a.forked_from
= b.id group by b.owner_id
union select count(distinct user_id), login from watchers,
projects, users where watchers.user_id
= projects.owner_id group by login;
Pattern: grouping / self-join / union-multiple-queries / 2-way-join

users following specific user
select count(follower_id), login from followers, users
where followers.follower_id
= users.id group by login;
Pattern: grouping / 2-way-join

specific user following other users
with c as (select count(users.id) as countUsers, follower_id as
followerID from followers, users where followers.follower_id =
users.id group by follower_id) select countUsers, login from c,
users where followerID = users.id;
Pattern: complex-query / grouping / 2-way-join

number of projects team owns
select count(projects.id), login from projects, users
where users.type = "ORG" group by login;
Pattern: grouping / string-search

Table 3.24 gives an overview of *all* Query-Patterns in the paper from Middleton et al. [2018]

Table 3.24: overview of *all* Query-Patterns in the paper from Middleton et al. [2018]

2-way-join: 14 times
distinct-search: 6 times
no-join: 1 times
string-search: 6 times
3-way-join: 4 times

grouping: 14 times
complex-query: 3 times
self-join: 2 times
using-NULL: 1 times
union-multiple-queries: 1 times

4. Discussion and Findings

In this Capital, we discuss the evaluated papers and present our findings. In the following Sections we discuss DBMS and data models, then we present our findings regarding the relevant attributes and SQL transformation possibility. This Capital ends with a Section on threats to validity.

4.1 DBMS and Data Models

During the evaluation of the papers, MySQL emerged as the most frequently used DBMS. This is because of the frequent usage of the GHTorrent dataset that comes with the MySQL dumps and ready-to-execute scripts for migrating CSV formatted data to the relational data model. Figure 4.1 gives an overview of the other DBMS in use. As can be concluded, for most of the research papers, no evidence of using a DBMS emerges. However, we advocate that in many of such cases, using a DBMS could have been proved efficient.

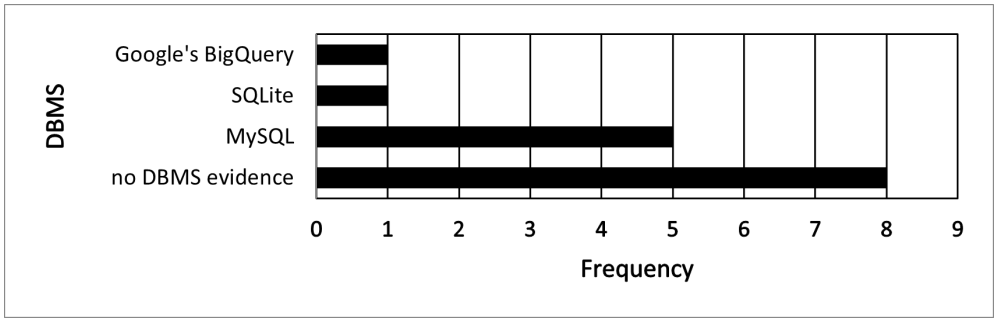

Figure 4.1: Frequency of the DBMS usage

The following describes some scenarios in which using a DBMS could have been proved beneficial. The recommendations for the ideal data management solutions have also been described in the following.

- In the paper *TNM: A Tool for Mining of Socio-Technical Data from Git Repositories* nested maps for storing developer and file pairs are used. These nested maps could have been stored in an object-oriented DBMS that allows nesting [ORD]. Additionally, the map of lists was used for commit hash as keys and changed lines as values. Such map of lists could have been stored efficiently using a type constructor in a DBMS such as in the Oracle DBMS [ORD]. JSON was also used for data storage — the usage of a DBMS such as MongoDB could have been efficient for JSON storage, as MongoDB stores JSON documents in BSON (a binary-encoded format) [JSO]. BSON is performant for encoding and decoding with various languages [JSO].
- In the paper *Escaping the Time Pit: Pitfalls and Guidelines for Using Time-Based Git Data* the authors use the Boa Infrastructure for identifying wrong timestamps. Even though the Boa Query language outputs results agnostic to a data model, the results could have been moulded to the relational data model and along with a relational DBMS, the further data processing could have been effective. For example, the authors used the command line tool grep for filtration purposes — this could have been replaced with the SQL for efficient processing [GRE]. Other than the relational data model, the graph data model could also have been used for storing the results of the Boa Infrastructure, because the resulting output tends to be varied and without a universal schema — this makes storing the unstructured results of the Boa Infrastructure in a Graph DBMS, an ideal decision. At last, the GitHub API and the Software Heritage was used for data collection in the JSON format. Same as recommended for the paper *TNM: A Tool for Mining of Socio-Technical Data from Git Repositories*, the authors here could have used MongoDB for the JSON formatted data.
- In the paper *Challenges in Developing Desktop Web Apps: A Study of Stack Overflow and GitHub* the authors crawl web pages for collecting data. During this process, they also apply filters such as data must contain a link to a GitHub repository. Such collection of data along with GitHub repository links could have been moulded into a knowledge graph that can be performant in executing queries in combination with a graph DBMS. Furthermore, authors mention collecting strongly intra-related data for 114,710 issue reports. As for instance, title, text, issue report author, etc. Such strongly intra-related data can be stored in the relational model or in the graph model that stores related data as neighbour nodes or property of nodes and edges. Nevertheless, authors mention no data model for storing these attributes' data.
- In the paper *A Study of Potential Code Borrowing and License Violations in Java Projects on GitHub* the authors work with data from 233,378 repositories. Whereby, they do not mention using any particular DBMS for the data management. In case of managing files as from 233,378 repositories, using a DBMS could have been an ideal decision. Furthermore, a complete version-control-system history of these repositories was gathered that consumed 1.55 TB on disk. Using a DBMS, potential redundancy could have been avoided — this could have reduced the 1.55 TB acquired space on the disk. As for instance, both the relational model and the graph data model offer possibilities for reducing redundancy.

- In the paper *Predicting Good Configurations for GitHub and Stack Overflow Topic Models* the authors gathered 5,000 documents for 8 selected programming languages. This calculates to be 40,000 documents in total. Nevertheless, authors do not mention using any DBMS for the data management. In such a case, a document oriented DBMS could have been efficient. Furthermore, the dataset was preprocessed before evaluation. For example, line breaks, HTML tags, comments, etc. were removed from the corpora. Although the author do not describe how this preprocessing was exactly carried out, such kind of preprocessing could have been efficient using a DBMS with a query language like SQL for the relational database. For JSON formatted data that the authors gathered, the same recommendation is to be made as in the discussion of the paper *TNM: A Tool for Mining of Socio-Technical Data from Git Repositories* of using MongoDB for storing JSON as BSON.
- For the paper *Identifying Experts in Software Libraries and Frameworks among GitHub Users* the recommendation for the JSON formatted data is the same as for the previously discussed papers, i.e. using MongoDB for storing JSON as BSON. The authors carry out filtration procedures on a list of 10,000 users, whereby they do not exactly describe if this filtration was carried out in a DBMS. Using a DBMS and a query language like SQL could have been efficient in their scenario. Furthermore, two sets were mapped to each other, i.e. the set of users and the set of GitHub accounts. This procedure could be represented as a variant of the bipartite graph. Hence, the graph data model along with a graph DBMS could have been used here for efficient processing. Nevertheless, no usage of such a data model (or the term bipartite graph) / DBMS occurs for the aforementioned use case. Additionally, during the final processing of data, missing values for certain GitHub attributes were replaced (based on specific criteria) with 0. For such a replacement of values, SQL could have been used if data were in the relational model. However, it is not specified how the aforementioned procedure was executed.

With regard to the used data models, the JSON and the relational data model emerged most frequently. This is because of the frequent usage of the GitHub API that results data in the JSON model and often usage of the GHTorrent dataset that comes with the relational schema. Figure 4.2 gives an overview of the other data models used. The graph data model also emerged, although not frequently. The reason being that few papers in our evaluation set used graph algorithms or graph data structure in their course of analysis.

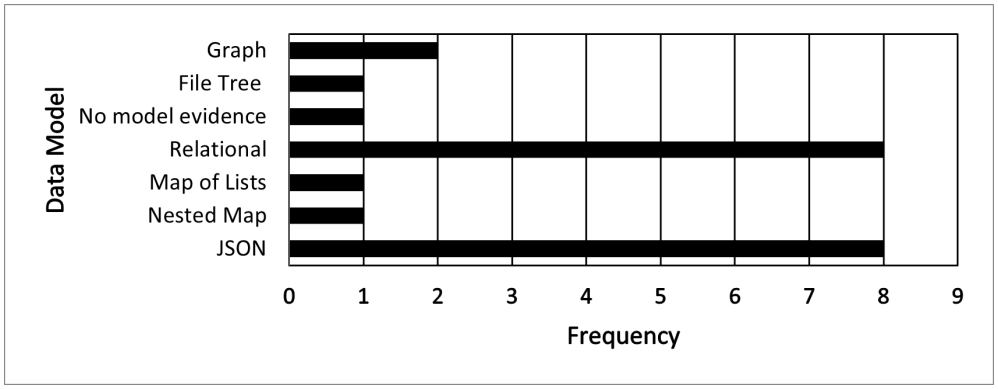

Figure 4.2: Frequency of the used data models

4.2 Relevant Attributes

The basic and computed attributes can be altogether categorized in the following dimensions: 1) user, 2) pull request, 3) commit, 4) projects, 5) files, 6) code, 7) issues.

Figure 4.3 represents the categorizations as dimensions.

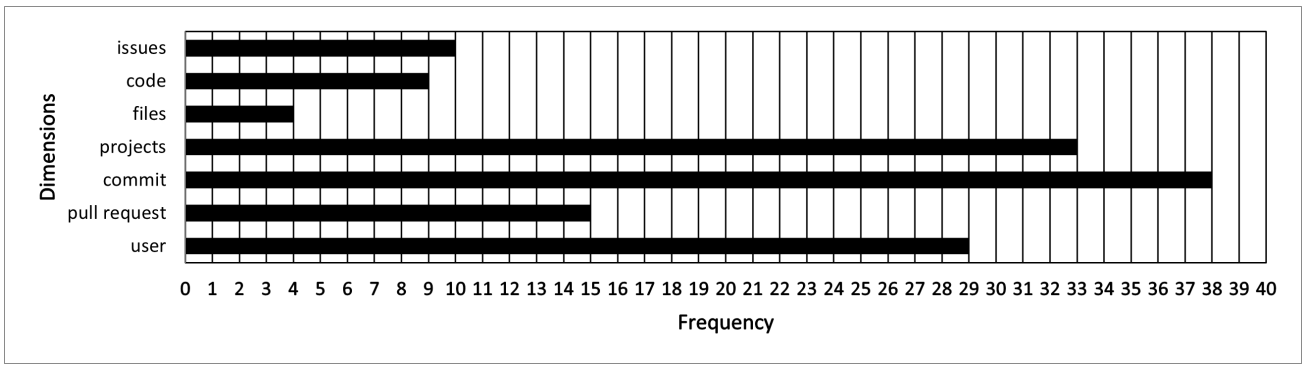

Figure 4.3: Frequency of basic and computed attributes categorized in dimensions

The following lists the tables that describe the respective dimension further.

- Table 4.1 concludes attributes categorized under *user*.
- Table 4.2 concludes attributes categorized under *pull request*.
- Table 4.3 concludes attributes categorized under *commit*.
- Table 4.4 concludes attributes categorized under *projects*.
- Table 4.5 concludes attributes categorized under *files*.
- Table 4.6 concludes attributes categorized under *code*.
- Table 4.7 concludes attributes categorized under *issues*.

Table 4.1: Summarized accessed attributes in dimension *User*

author name
user name
contributors
user profile page URL
user email
if author owns repo
if author has write access
if author contributed
if author is follower
author date
author email
author timezone
committer date
committer email
committer name
committer timezone
number of unique active developers
users in multiple repository
users participating in multiple language repositories
number of committer per repository
if user has write access
number of comments user adds in issue discussions
tenure (time between user interaction / dataset snapshot)
users following specific user
specific user following other users

Table 4.2: Summarized accessed attributes in dimension *pull request*

if user initiated pull request
pull request discussions
if user merging / closing pull request
if user committing without pull request
number of pull requests opened
comments to pull requests created by user
comments to pull requests by user created by others
merged/non-merged pull requests
comments on merged/non-merged pull requests
time-to-merge/time-to-close pull requests
commits of merged/non-merged pull requests
time since the first pull request
number of pull request authors
number of pull requests opened
pull requests

Table 4.3: Summarized accessed attributes in dimension *commit*

commit Messages
parent Commits
number of commits
commit hash
commit timestamp
number of commits (on basis author)
if commit merged
commit message length
commits
commitsClientFiles
commitsImportLibrary
daysSinceFirstImport
daysSinceLastImport
daysBetweenImports
avgDaysCommitsClientFiles
avgDaysCommitsImportLibrary
number of daily commits
commits made by users for each language
number of commits per user
number of commits per repository
number commit comments by user
number of commits user made to own projects
if commit in main branch
name of branch commit to
number of modifications in commit

Table 4.4: Summarized accessed attributes in dimension *projects*

project name
project URL
stars for projects
watchers per project
project's license
project's description
project's GitHub page
project's size
forks
projects
projectsImport
repository badges
number of repositories per user
project team roles
number of team projects user watches
number of projects user forks
number of non-forked projects user created
number of projects team owns
project clone URL
number of users watched OR forked project from specific user

Table 4.5: Summarized accessed attributes in dimension *files*

changed files name
number of files
project's ReadMe file
continuous integration (CI)

Table 4.6: Summarized accessed attributes in dimension *code*

code language
lines of code added
lines of code removed
codeChurn
codeChurnClientFiles
imports
code language
number of languages per user
code language

Table 4.7: Summarized accessed attributes in dimension *issues*

issue reports
issue reports title
text on issue discussion page
issue reports author
issue reports labels
issue reports current status
issue reports creation / submission timestamp
issue reports last edit / closing timestamp
issues
number of issues user opens in projects

4.2.1 Synopsis for relevant attributes

Figure 4.4 represents the frequency of accessed basic attributes and Figure 4.5 represents the frequency of accessed computed attributes.

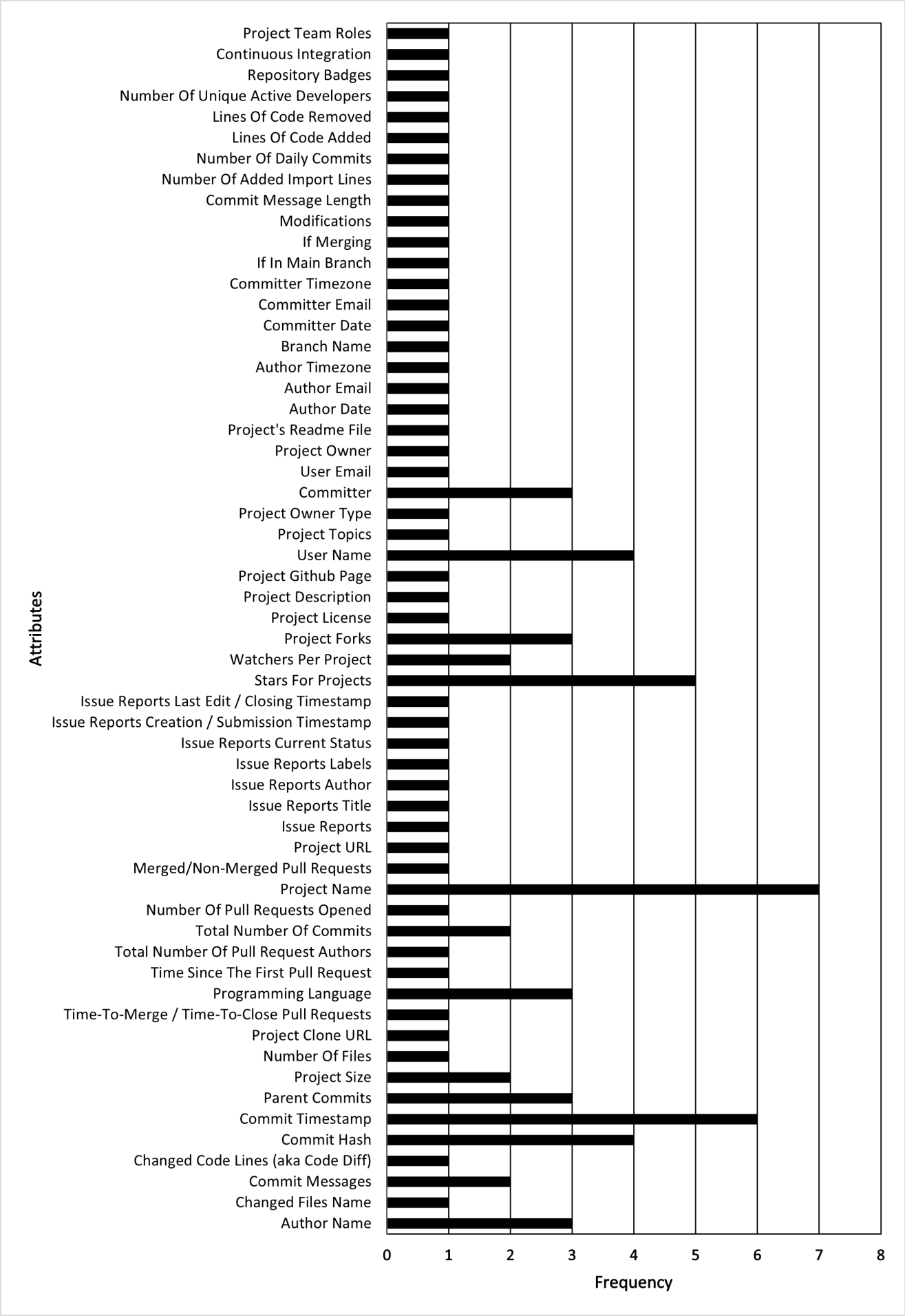

Figure 4.4: Frequency of accessed basic attributes

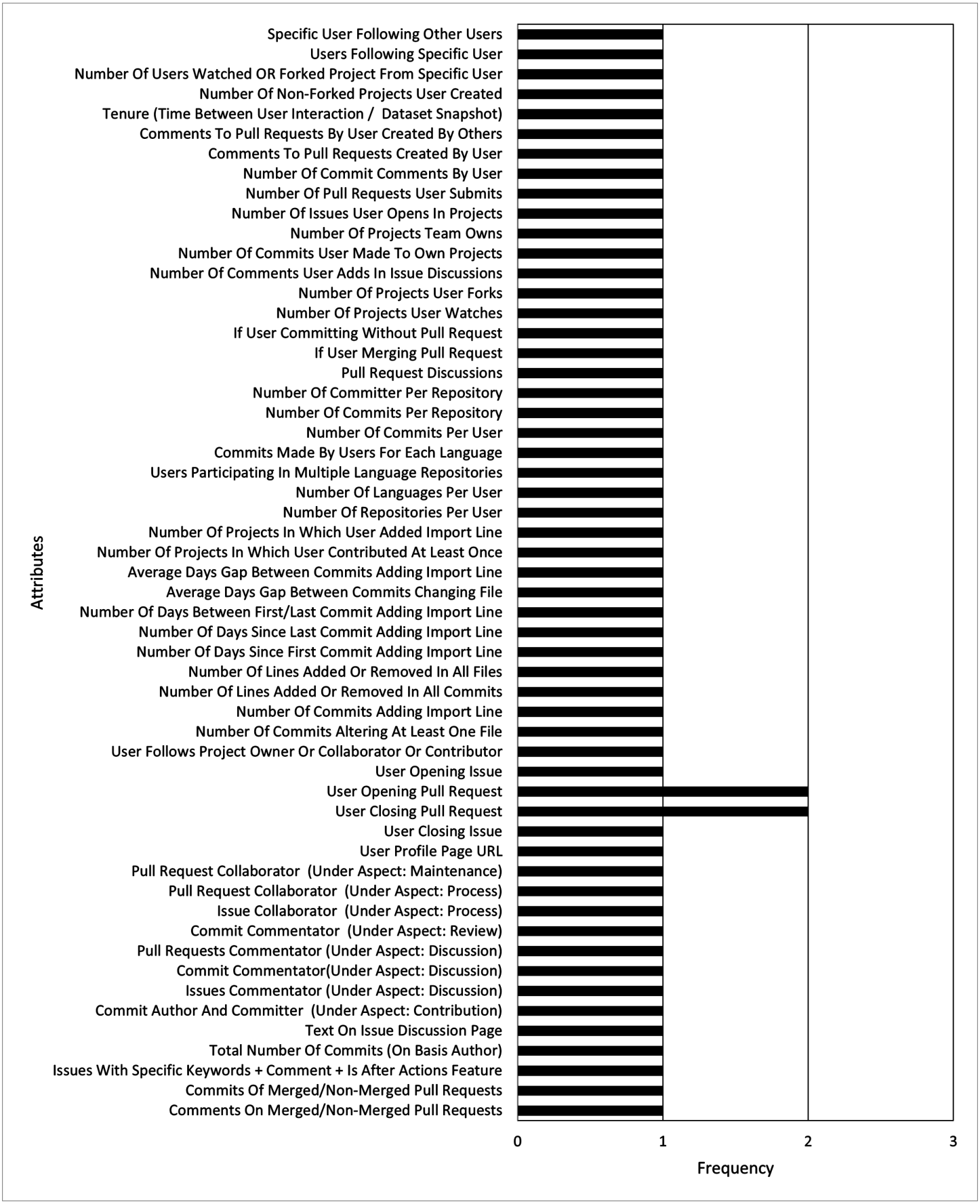

Figure 4.5: Frequency of accessed computed attributes

The computed attributes can be divided into their basic attributes counterparts. This insight can be derived from Table 4.8 and the Sankey Diagram 4.6.

Table 4.8: Computed attributes divided into their basic attributes counterparts

Computed:	Comments On Merged/Non-Merged Pull Requests
Basic parts:	(1) Pull Request Comment, (2) Pull Request Status
Computed:	Commits Of Merged/Non-Merged Pull Requests
Basic parts:	(1) Commit Hash, (2) Pull Request Status
Computed:	Issues With Specific Keywords + Comment + Is After Actions Feature
Basic parts:	(1) Issue Timestamp, (2) Issue Title, (3) Issue Comments
Computed:	Total Number Of Commits (On Basis Author)
Basic parts:	(1) Commit Hash, (2) Author Id
Computed:	Text On Issue Discussion Page
Basic parts:	(1) Issue Id, (2) Issue Discussion Page, (3) Issue Discussion Page Text
Computed:	Commit Author And Committer (Under Aspect: Contribution)
Basic parts:	(1) Author Id, (2) Committer Id
Computed:	Issues Commentator (Under Aspect: Discussion)
Basic parts:	(1) Issue Id, (2) Commentator
Computed:	Commit Commentator(Under Aspect: Discussion)
Basic parts:	(1) Commit Hash, (2) Commentator
Computed:	Pull Requests Commentator (Under Aspect: Discussion)
Basic parts:	(1) Pull Request Id, (2) Commentator
Computed:	Commit Commentator (Under Aspect: Review)
Basic parts:	(1) Commit Hash, (2) Commentator
Computed:	Issue Collaborator (Under Aspect: Process)
Basic parts:	(1) Issue Action, (2) Issue Actor, (3) Issue Id
Computed:	Pull Request Collaborator (Under Aspect: Process)
Basic parts:	(1) Pull Request Id, (2) Pull Request Opener, (3) Pull Request Actor
Computed:	Pull Request Collaborator (Under Aspect: Maintenance)
Basic parts:	(1) Pull Request Id, (2) Pull Request Opener, (3) Pull Request Actor
Computed:	User Profile Page URL
Basic parts:	(1) User Id, (2) Profile Page URL

Computed: User Closing Issue
Basic parts: (1) User Id, (2) Issue Action

Computed: User Closing Pull Request
Basic parts: (1) User Id, (2) Pull Request Action

Computed: User Opening Pull Request
Basic parts: (1) User Id, (2) Pull Request Action

Computed: User Opening Issue
Basic parts: (1) User Id, (2) Issue Action

Computed: User Follows Project Owner
Or Collaborator (in context) Or Contributor (in context)
Basic parts: (1) User Id, (2) Project Owner, (3) Committer,
(4) Issue Action, (5) Pull Request Action

Computed: Number Of Commits Altering At Least One File
Basic parts: (1) Commit Hash, (2) Files

Computed: Number Of Commits Adding Import Line
Basic parts: (1) Commit Hash, (2) Code

Computed: Number Of Lines Added Or Removed In All Commits
Basic parts: (1) Code, (2) Commit Hash

Computed: Number Of Lines Added Or Removed In All Files
Basic parts: (1) Code, (2) Files

Computed: Number Of Days Since First Commit Adding Import Line
Basic parts: (1) Commit Timestamp, (2) Code

Computed: Number Of Days Since Last Commit Adding Import Line
Basic parts: (1) Commit Timestamp, (2) Code

Computed: Number Of Days Between First/Last Commit Adding Import Line
Basic parts: (1) Commit Timestamp, (2) Code

Computed: Average Days Gap Between Commits Changing File
Basic parts: (1) Commit Timestamp, (2) Files

Computed: Average Days Gap Between Commits Adding Import Line
Basic parts: (1) Commit Timestamp, (2) Code

Computed: Number Of Projects In Which User Contributed At Least Once
Basic parts: (1) Project Id, (2) User Id, (3) Committer Issue Action, (4) Pull Request Action

Computed: Number Of Projects In Which User Added Import Line
Basic parts: (1) Project Id, (2) User Id, (3) Code

Computed: Number Of Repositories Per User
Basic parts: (1) Project Id, (2) User Id

Computed: Number Of Languages Per User
Basic parts: (1) Programming Language, (2) User Id

Computed: Users Participating In Multiple Language Repositories
Basic parts: (1) User Id, (2) Programming Language, (3) Project Id

Computed: Commits Made By Users For Each Language
Basic parts: (1) Commit Hash, (2) User Id, (3) Programming Language

Computed: Number Of Commits Per User
Basic parts: (1) Commit Hash, (2) User Id

Computed: Number Of Commits Per Repository
Basic parts: (1) Commit Hash, (2) Project Id

Computed: Number Of Committer Per Repository
Basic parts: (1) Committer, (2) Project Id

Computed: Pull Request Discussions
Basic parts: (1) Pull Request Id, (2) Discussion Text

Computed: If User Merging Pull Request
Basic parts: (1) User Id, (2) Pull Request Action

Computed: If User Committing Without Pull Request
Basic parts: (1) Committer, (2) Project Id, (3) Pull Request Action

Computed: Number Of Projects User Watches
Basic parts: (1) User Id, (2) Project Id, (3) User Action

Computed: Number Of Projects User Forks
Basic parts: (1) Project Id, (2) Forked From, (3) User Id

Computed: Number Of Comments User Adds In Issue Discussions
Basic parts: (1) Issue Comments, (2) User Id

Computed: Number Of Commits User Made To Own Projects
Basic parts: (1) Commit Hash, (2) Committer, (3) Project Id

Computed: Number Of Projects Team Owns
Basic parts: (1) Project Id, (2) User Type

Computed: Number Of Issues User Opens In Projects
Basic parts: (1) Issue Id, (2) User Id, (3) Issue Action, (4) Project Id

Computed: Number Of Pull Requests User Submits
Basic parts: (1) Pull Request Id, (2) Pull Request Action, (3) User Id

Computed: Number Of Commit Comments By User
Basic parts: (1) Commit Hash, (2) Commit Comment, (3) User Id

Computed: Comments To Pull Requests Created By User
Basic parts: (1) Pull Request Comment, (2) User Id, (3) Pull Request Action

Computed: Comments To Pull Requests By User Created By Others
Basic parts: (1) Pull Request Comment, (2) User Id, (3) Pull Request Action

Computed: Tenure (Time Between User Interaction / Dataset Snapshot)
Basic parts: (1) User Action, (2) User Id, (3) Dataset Timestamp

Computed: Number Of Non-Forked Projects User Created
Basic parts: (1) Project Id, (2) User Id, (3) User Action, (4) Forked From

Computed: Number Of Users Watched OR Forked Project From Specific User
Basic parts: (1) User Id, (2) User Action, (3) Project Id

Computed: Users Following Specific User
Basic parts: (1) User Id, (2) User Action

Computed: Specific User Following Other Users
Basic parts: (1) User Id, (2) User Action

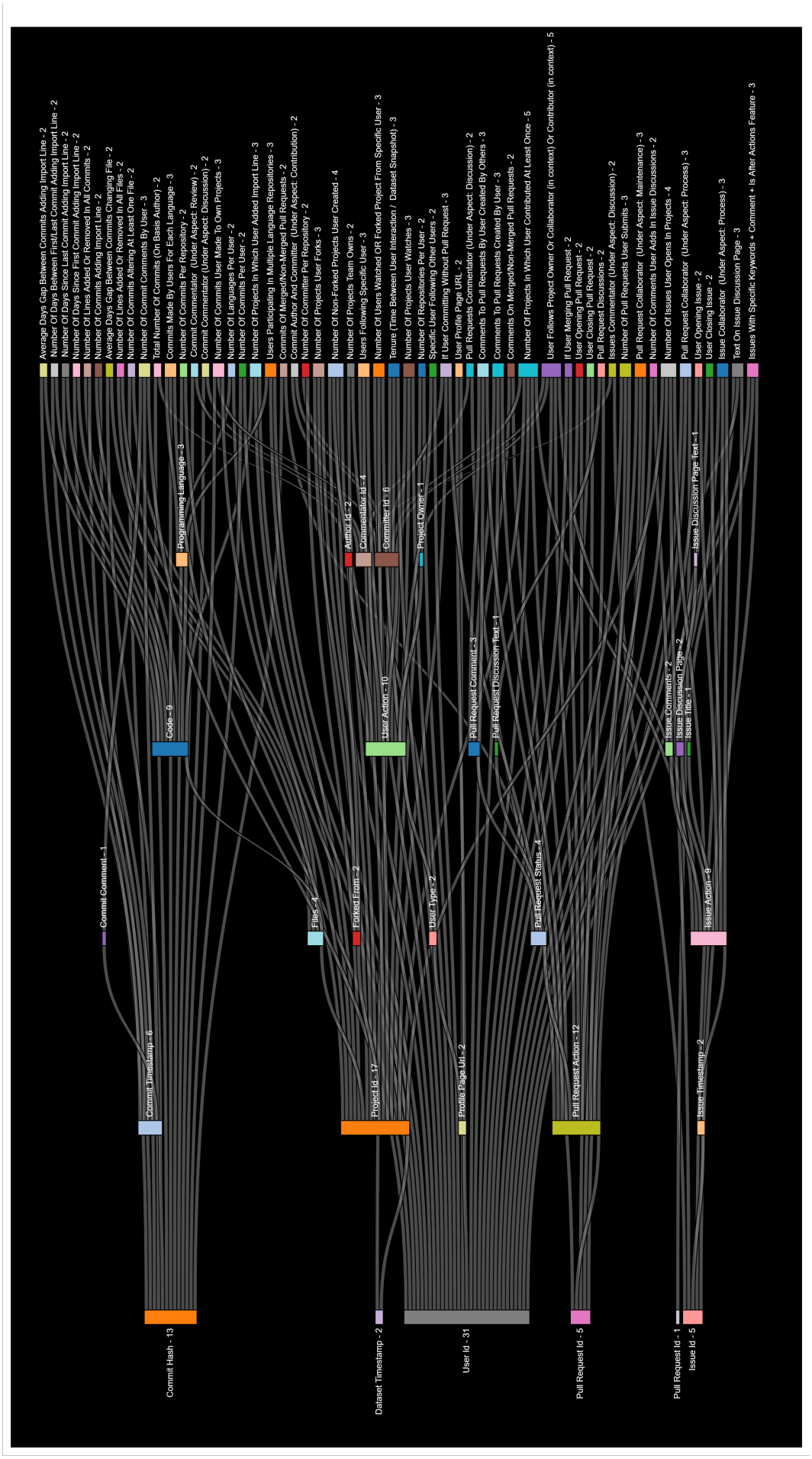

Figure 4.6: Sankey Diagram representing basic attributes at bottom and computed attributes on top (with hierarchies in between)

4.3 SQL Transformation

The Extract, Transform, Load (ETL) cost for all evaluated papers' data is estimated on basis of 1) the data model(s) already used in the paper, 2) how data is managed, 3) how data management is described, i.e. its documentation 4) how well is the data schema already defined, 5) how accessible is data for us to work with it.

Depending on these factors, the cost was divided into two categories, 1) high, 2) average. Figure 4.7 gives an overview of these categories.

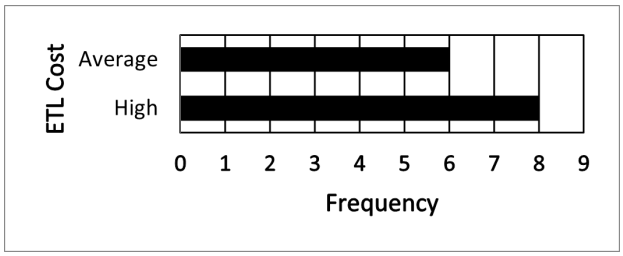

Figure 4.7: ETL cost for all evaluated papers' data

4.4 Threats to validity

In this section the threats to the validity of our evaluation and results are described and discussed.

- For collecting papers during the literature research, only the title of the paper containing the term git was considered. However, there may be papers that do not mention git in the title but still access GitHub attributes in their research course. Hence, our literature research methodology is sound, but not complete. Furthermore, only the MSR conference papers were filtered for relevant papers, whereby there exist other conferences and journals that include papers working with the GitHub data. These are, for example, conferences like EASE, OSS, SAC, SEKE and Journals like ESE, IST, JSS, SCP [Cosentino et al. [2016]].
- During the readings of the selected papers, only the content of the paper and the general overview of external resources provided by the authors were considered to derive conclusions about the used data models or the DBMS. Nevertheless, presentation slides related to these papers or in depth reading of the external sources' documentations might have revealed more information on the used data models or the DBMS. Hence, the derived conclusions in the aforementioned scenario might be wrong.
- The derivation of the atomic attributes from the computed ones is disputable. An an instance, if a user is following another user on GitHub, then it can be expressed in a variety of combinations of GitHub attributes, i.e. 1) *user id, user action*, 2) *user id, if_user_follow*, 3) *user id, follow_from, follow_to*.

  This is mainly because the authors of the respected papers might name the attributes differently or present the attributes variously fitting to their study context. How attributes are managed in the tables and the schema that the tables build together also play a role in the symbolic names of the attributes.

5. Related Work

In this Capital, we present related work. The following Sections describe data management and data retrieval strategies.

5.1 Data Management Strategies

In this section, we present the related work concerning data management strategies for GitHub data. Table 5.1 represents an overview of such significant work.

Table 5.1: Existing work concerning data management for GitHub data

Public Git Archive [PGA]
Google BigQuery Cloud Service [GBC]
Software Heritage [SH]
GitHub Archive [GHA]
GHTorrent [GHT]

The aforementioned work is related, however not taken into account for our study, because of the similarity and the difference between the related work and our thesis. The similarity is that our thesis and the related work try to support researchers in data management. The difference is that the related work provide data packed in a data model, and our thesis observes the *de facto* usage of data models and opens discussion about other data models.

5.1.1 Public Git Archive

Public Git Archive (PGA) was introduced in the year 2018 by Markovtsev and Long [2018]. The size of PGA dataset, as authors mention in their paper, is 3 TB. The dataset itself is based on the GHTorrent metadata. This means that the GHTorrent metadata is used for defining an initial list of repositories. This list gets used in combination with the Siva file archival service. Siva is an archival format similar to zip and allows runtime of constant time random file access.

An index providing an overview of repository files can be generated for the files archived in the Siva format. This index content is in the CSV format with columns such as URL, siva_filenames, licence, commits_count, etc. This CSV file content can be easily transported to the relation data model. Nevertheless, other than the CSV content, the GitHub data itself is not in any particular data model.

5.1.2 Google BigQuery Cloud Service

Under the name BigQuery, Google provides its service for data management and data analysis on a serverless infrastructure that is scalable and distributed. It is a paid service, with free trial options having limitations.

The main datasets stored in BigQuery's server are from the GHArchive, and the GHTorrent projects. The GHArchive's dataset updates are planned on hourly basis and the GHTorrent's dataset updates are planned monthly. However, the last GHTorrent dataset available on the BigQuery's server dates back to June 2019 [TWI]. This is because the update of the GHTorrent dataset on BigQuery is dependent on the GHTorrent's project owner [GHO].

It has to be explicitly mentioned that instead of an own dataset composition, the BigQuery service only provides the datasets of other projects for querying. Despite this, BigQuery lands as a separate work in our list of related work, because BigQuery allows users to join the aforementioned available datasets with other datasets such as TravisTorrent, Reddit, etc. [BJD].

5.1.3 Software Heritage

Software Heritage service was introduced in 2017. The goal of the service is quite novel: code should never get lost. Anyone can search for code content of GitHub repositories, using filters such as project name, in the Software Heritage's search engine and download the repository's content as zip. Keeping deduplication as focus, the underlying data model for the storage has been chosen. This results in the Merkle Direct Acyclic Graph (DAG) as the data model, because each hash identifier for nodes that is generated using the content of a node, is unique and enables detection of duplicate additions in the dataset. In the aforementioned statement, entities such as blobs, commits, tags, directories, etc. are represented by nodes. It is important to mention that the DAG data structure is not to be mistakenly confused with the tree data structure in the Software Heritage scenario, because file contents can be forked and merged back — resulting in the graph data structure. In 2019 Pietri et al. [2019] introduced their paper in the MSR conference concerning the Software Heritage DAG dataset as downloadable CSV dumps, Apache Parquert files, and as a ready-to-use instance on Amazon Athena.

5.1.4 GitHub Archive

GHArchive is another initiative for recording, archiving, and making accessible the data from GitHub. The data recordings start from the year 2011 till today. GHArchive aggregates 20+ event types provided by GitHub, such as, new commits, comments, fork events, opening new pull requests, etc. The aggregation of all these

events occurs on an hourly basis that is accessible through an HTTP client such as WGET. The data itself is available as JSON encoded — same as the GitHub API provides it. This means that the researchers using the GHArchive need to work with the raw data that can be manually processed to fit in a data schema. The GHArchive project's data is also assessable through Google's BigQuery. The dataset version available on BigQuery is also updated on an hourly basis and allows execution of SQL queries — this implies that the dataset version on BigQuery is in the relational data model. Nevertheless, the relational model schema is divided *only* in three parts, i.e. year, month, and day. Hence, the dataset's schema organization on BigQuery follows only the temporal factor. In case of the GHArchieve we find the time dimension for the *schema organisation* not so important, an ideal schema example can be derived from the schema of the relational model in the GHTorrent project.

Figure 5.1 represents an example query from the day division with the exact date as the table name.

```
SELECT type, repo.name, actor.login,
FROM `githubarchive.day.20160101`
WHERE type = 'IssuesEvent'
```

Figure 5.1: Example query for GHArchive dataset

5.1.5 GHTorrent

The GHtorrent initiative is the most successful one among the related works presented in this chapter [GHT]. The project was introduced in 2013 by Gousios Georgios, whereby the dataset included data from GitHub dated since February 2012. There exist two parts of the project, 1) storing raw JSON data to MongoDB, 2) extracting data's structure and storing it to MySQL database. As of 2020 January, the MongoDB dataset comprises 18TB of compressed data as JSON. The MySQL dataset comprises 6.5 billion rows of metadata that is structured in many tables in a well-defined relational data model. As an example, there exist tables for user, projects, pull requests, issue, etc.

The GHTorrent data can be downloaded via HTTP, however in the initial phase of the project, the data was available through Bittorrent. Hence the name of the project includes *torrent* as suffix [GHT].

5.2 Data Retrieval Strategies

There exist further services for efficient collection of GitHub data. Nevertheless, these services are not to be considered as related to data management strategies but instead as data retrieval. A list of these services is to be abstracted from Table 5.2

Table 5.2: Existing work concerning data retrieval for GitHub data

GHCrawler [GHC]
BOA [BOA]
Github's official Octokit [GOO]
Github Search API [GSA]
Github REST API [GRA]
Kibble [KIB]
SEART GitHub Search [SGS]

6. Conclusion

In this work, we investigated papers from the Mining Software Repositories (MSR) conference to define GitHub attributes relevant to researchers and defined the data models and the DBMS that are used by researchers in their analysis. Additionally, we estimated if it is possible to transform the researchers' analysis in SQL queries to define the evaluation from the view point of the relational model.

Our findings demonstrate that attributes providing information about commits are most relevant. After that, attributes revealing information about repositories, such as number of files, hold priority. The third most relevant attributes are those that give information about the user.

With respect to the DBMS and the data models, we have discovered that JSON and the relational data model both emerge with the frequency of 36.36%. The graph data model shows up only at 9.09%. The contrast of the relational and the graph data model results to be of 300%. Frequency of all other data models results 4.54% each. No DBMS usage accounts to be at 53.33% frequency, MySQL at 33.33%, Google's BigQuery and SQLite both at 6.66% each.

Concerning the evaluation from the view point of the relational model, our findings conclude that for all papers the transformation of the researcher's evaluation in SQL queries is *limitedly* possible, because of lacking schema or non-accessible data. The estimated Extract, Transform, and Load (ETL) cost for 57% of the papers is high and for 42% average.

6.1 Synopsis

Summarizing all the findings, we can say that the big data researchers collect from GitHub mostly lacks proper data management. However, the transit of data management for software engineering research analysis to a DBMS is limitedly possible. The results inform that there are cases where the relational data model emerges, albeit mostly without well-defined schema. On the other side, the use of the other popular data models, such as the graph data model, rarely comes in the

focus. The aforementioned results are especially interesting, because we have also discovered 1) commit, 2) repository, and 3) user related attributes to be the most relevant for the software engineering research questions that fit well in context of a *forest* (graph data model).

Bibliography

ACT. Acceptance rate MSR. https://dl.acm.org/doi/proceedings/10.1145/3379597#acceptance-rates. (cited on Page 4)

AWS. Amazon Web Services. https://aws.amazon.com/nosql/document/. (cited on Page 15)

BJD. BigQuery Join Datasets. https://ghtorrent.org/gcloud.html. (cited on Page 52)

BOA. BOA Language. http://boa.cs.iastate.edu/boa/?q=boa/job/public/545. (cited on Page vii, 12, and 54)

Patrick Eric Carlson. *Engaging developers in open source software projects: harnessing social and technical data mining to improve software development*. PhD thesis, Iowa State University, 2015. (cited on Page 6)

CMC. Coin Market Cap. https://coinmarketcap.com. (cited on Page 27)

Eldan Cohen and Mariano P Consens. Large-scale analysis of the co-commit patterns of the active developers in github's top repositories. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 426–436. IEEE, 2018. (cited on Page x, 28, and 29)

Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968. (cited on Page 7)

Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Findings from github: methods, datasets and limitations. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 137–141. IEEE, 2016. (cited on Page 50)

DAT. Data Showcase papers. https://2021.msrconf.org/track/msr-2021-data-showcase?#About. (cited on Page 5)

Hongbo Fang, Daniel Klug, Hemank Lamba, James Herbsleb, and Bogdan Vasilescu. Need for tweet: How open source developers talk about their github work on twitter. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 322–326, 2020. (cited on Page ix, 20, and 21)

Samuel W Flint, Jigyasa Chauhan, and Robert Dyer. Escaping the time pit: Pitfalls and guidelines for using time-based git data. *arXiv preprint arXiv:2103.11339*, 2021. (cited on Page ix, 4, 11, 12, and 13)

GBC. Google BigQuery Cloud Service. https://console.cloud.google.com/marketplace/product/github/github-repos. (cited on Page 51)

GHA. GitHub Archive. https://www.gharchive.org/. (cited on Page 51)

GHC. GHCodeSnippetHistory. https://github.com/manessaraj/GHCodeSnippetHistory. (cited on Page 54)

GHO. BigQuery On GitHub. https://github.com/fhoffa/analyzing_github. (cited on Page 52)

GHT. GHTorrent. https://ghtorrent.org/. (cited on Page 51 and 53)

Yaroslav Golubev, Maria Eliseeva, Nikita Povarov, and Timofey Bryksin. A study of potential code borrowing and license violations in java projects on github. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 54–64, 2020. (cited on Page ix, 14, and 16)

Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 431–442, 2020. (cited on Page ix, 16, 17, 18, and 20)

GOO. Github's official Octokit. https://docs.github.com/en/rest/overview/libraries. (cited on Page 54)

Christoph Gote, Ingo Scholtes, and Frank Schweitzer. git2net-mining time-stamped co-editing networks from large git repositories. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 433–444. IEEE, 2019. (cited on Page x, 23, and 24)

GRA. GitHub REST API. https://docs.github.com/en/rest. (cited on Page 54)

GRE. GREP or SQL. https://stackoverflow.com/questions/5927241/what-is-generally-faster-grepping-through-files-or-running-a-sql-like-x-query/5927291. (cited on Page 36)

GSA. GitHub Search API. https://docs.github.com/en/rest/reference/search. (cited on Page 54)

JSO. JSON Databases. https://www.quackit.com/json/tutorial/list_of_json_databases.cfm. (cited on Page 36)

KIB. Kibble. https://kibble.apache.org. (cited on Page 54)

Timothy Kinsman, Mairieli Wessel, Marco A Gerosa, and Christoph Treude. How do software developers use github actions to automate their workflows? *arXiv preprint arXiv:2103.12224*, 2021. (cited on Page ix, 10, and 11)

Saraj Singh Manes and Olga Baysal. Studying the change histories of stack overflow and github snippets. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 283–294. IEEE, 2021. (cited on Page ix, 1, 8, and 9)

Vadim Markovtsev and Waren Long. Public git archive: a big code dataset for all. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 34–37, 2018. (cited on Page 15 and 51)

Justin Middleton, Emerson Murphy-Hill, Demetrius Green, Adam Meade, Roger Mayer, David White, and Steve McDonald. Which contributions predict whether developers are accepted into github teams. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 403–413. IEEE, 2018. (cited on Page x, 1, 30, 31, and 33)

MIN. Minig Challenge Papers. https://2021.msrconf.org/track/msr-2021-mining-challenge. (cited on Page 5)

Joao Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. Identifying experts in software libraries and frameworks among github users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 276–287. IEEE, 2019. (cited on Page x, 24, 25, and 26)

MSR. Mining Software Repositories. http://www.msrconf.org/. (cited on Page 4)

Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6): 3219–3253, 2017. (cited on Page 10)

ORD. Object Relational Databases. https://en.wikipedia.org/wiki/Object%E2%80%93relational_database. (cited on Page 36)

PGA. Public Git Archive. https://github.com/src-d/datasets/tree/master/PublicGitArchive. (cited on Page 51)

Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The software heritage graph dataset: public software development under one roof. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 138–142. IEEE, 2019. (cited on Page 52)

Gian Luca Scoccia, Patrizio Migliarini, and Marco Autili. Challenges in developing desktop web apps: a study of stack overflow and github. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 271–282. IEEE, 2021. (cited on Page ix, 1, 13, and 14)

SGS. SEART GitHub Search. https://seart-ghs.si.usi.ch/. (cited on Page 54)

SH. Software Heritage. https://www.softwareheritage.org/. (cited on Page 51)

Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press. ISBN 9781450355735. doi: 10.1145/3236024.3264598. URL http://dl.acm.org/citation.cfm?doid=3236024.3264598. (cited on Page 23)

Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis, 2015. (cited on Page 28)

Sho Suzuki, Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. An application of the pagerank algorithm to commit evaluation on git repository. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 380–383. IEEE, 2017. (cited on Page 7)

Nikolai Sviridov, Mikhail Evtikhiev, and Vladimir Kovalenko. Tnm: A tool for mining of socio-technical data from git repositories. *arXiv preprint arXiv:2103.09766*, 2021. (cited on Page ix, 6, and 8)

SWH. Software Heritage. https://www.softwareheritage.org/. (cited on Page 12)

TEC. Technical Papers. https://students.unimelb.edu.au/academic-skills/explore-our-resources/report-writing/technical-report-writing. (cited on Page 5)

Christoph Treude and Markus Wagner. Predicting good configurations for github and stack overflow topic models. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 84–95. IEEE, 2019. (cited on Page ix, 22, and 23)

Asher Trockman, Rijnard van Tonder, and Bogdan Vasilescu. Striking gold in software repositories? an econometric study of cryptocurrencies on github. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 181–185. IEEE, 2019. (cited on Page x, 26, and 27)

TWI. GHTorrent On Twitter. https://twitter.com/ghtorrent/status/1222529377629605889?cxt=HHwWgsC8kbz_pfchAAAA. (cited on Page 52)

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

Magdeburg, 4. March 2022