Exploring the potential of semantics on the git protocol and its extensions

Masterarbeit zur Erlangung des Grades

Master of Science (M.Sc.)

im Studiengang Computer Science

an der Rheinischen Friedrich-Wilhelms-Universität Bonn

Erstbetreuer: Prof. Dr. Jens Lehmann
Zweitbetreuer: Dr. Damien Graux

vorgelegt im Juli 2019 von:

Matthias Böckmann
Matrikelnummer: 2536943

Dennis Oliver Kubitza
Matrikelnummer: 2563042

Abstract

Many programmers use Version Control Systems (VCS) such as `git` to organise their collaboration with other developers. GitHub is the largest `git` repository hoster on the web, providing a wealth of information about open source software projects and social interactions of their developers. In this thesis we describe, analyse and utilise the SemanGit dataset. This semantic resource encapsulates data generated by billions of executions of the `git` protocol via GitHub, alongside user and project data of over 30 million GitHub accounts and 100 million projects, summing up to over 24 billion relations. We elaborate the difficulties with loading and querying a dataset of this size. Afterwards, we discuss how SemanGit can be utilised in theory and practice and manage to run first analyses by the provision of a SPARQL endpoint. We perform several sample analyses, such as finding trends in programming languages, finding that there are strong differences in the usage of programming languages when comparing geographically distant regions or projects of different popularity. We also uncover new information about international cooperation and investigate if the social structure of organisations has an impact on their success. Lastly, we describe how more complex analyses can be performed by interlinking our dataset with others, such as DBpedia.

Ich versichere hiermit, dass ich die in der vorstehenden Masterarbeit mit meinem Zeichen gekennzeichneten Teile selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass die vorgelegte Arbeit noch an keiner anderen Hochschule zur Prüfung vorgelegt wurde und dass sie weder ganz noch in Teilen bereits veröffentlicht wurde. Wörtliche Zitate und Stellen, die anderen Werken dem Sinn nach entnommen sind, habe ich in jedem einzelnen Fall kenntlich gemacht.

Ort, Datum	Name	Unterschrift
Ort, Datum	Name	Unterschrift

Acknowledgements

We would like to thank Professor Jens Lehmann for his supervision and Damien Graux for his continuous support. Their doors were always open to us and they provided us with great feedback and suggestions during every stage of the work. We thank everyone who supported us along the way. Both of us would like to say thank you to our families for morally supporting us over many years of studying!

As this thesis was authored by two persons, we annotate each section.
The content after captions marked with (†) has been written by Dennis Oliver Kubitza.
The content after captions marked with (*) has been written by Matthias Böckmann.

Table of Contents

List of Figures

List of Tables

1 (†) Introduction

The growing interest in free and open-source software which occurred over the last decades has accelerated the usage of so-called Version Control Systems (VCS), which allow developers to collaborate with each other more easily. These tools track and save any changes made to the code, so that one can revert to prior versions at will. Besides providing a history of the development process of the code, they also enable multiple authors to manage and merge their contributions into one single version. If conflicting changes are made, such as two agents editing the same line of code, the VCS reports an error during the merging process and developers can review and choose whose version of the conflicting part to accept. This is particularly useful, if authors work at different times or across geographical distances. One of the most popular VCS tool suits is `git` [1], which is free, open-source and supports a wide range of operating systems. It is even provided out of the box in several popular integrated development environments (IDE).

Source code changes are summarised into a commit and can be pushed to `git` repositories, which are the core of the system. Such repositories can be self-hosted for full control over the data flow, or one can choose one of the many online `git`repository hosts. These providers take care of server management, the `git` setup and in many cases provide simple interfaces for the `git` protocol, bridging the necessity of executing protocol commands in the command line. Besides the benefits of a decentralised server structure, these online providers usually also offer additional features that are not part of the `git` protocol. For example, GitHub, the currently largest `git` repository hoster on the web [2], offers an issue tracking system, where users can make suggestions or report bugs. With its increasing number of users, it also servers as social platform for developers, adopting features of popular social networks like following other users or watching projects.

With GitHub reaching 36 million users and 100 million projects as of April 2019 [3], there is abundant information about code evolution and social interactions between developers available. This makes GitHub an excellent data source for social and economic analysis and has already attracted a lot of attention from researchers [4]. This data can be queried via the `GitHub API`, but mining capabilities are restricted through an hourly query limit. GHTorrent [5] bypasses this limit by mining the API with hundreds of community donated authentication tokens over the last years, providing data dumps. However, to the best of our knowledge, nobody has attempted to collect and analyse this data in a graph data model. This alternative representation is well suited for typical

graph traversal problems, such as finding followers of followers.

In this master thesis, we present a large scale graph database filled with information extracted from the execution of the `git` protocol, collected from the users of GitHub and their social interactions on the platform. We decided to name this project "SemanGit", which is a neologism from "semantic" and "git", emphasising the used technology of enriching the extracted data with semantics. These semantics of the data are summarised in an ontology, which was created with the WebVOWL [6] tool. It defines the structure of classes and relations that occur in the SemanGit data with expressions from the Web Ontology Language (OWL) [7] and the Resource Description Framework Schema (RDFS) [8]. Utilising this ontology, we create a Resource Description Framework (RDF) graph [9]. In RDF, facts are stored as (Subject, Predicate, Object) triples, indicating that the nodes *Subject* and *Object* share the relation *Predicate*, displayed by an edge of type *Predicate* between the two nodes. The motivation behind RDF is described by the World Wide Web Consortium (W3C) as follows [10]:

> *[...] a common framework that allows data to be shared and reused across application, enterprise, and community boundaries, to be processed automatically by tools as well as manually, including revealing possible new relationships among pieces of data.*

In fact, the usage of RDF for SemanGit might enable a future interlinking with additional data sources like *LinkedIn, StackOverflow* or other `git` providers. Note that the most recent version of the dataset is already interlinked with DBpedia [11], one of the most referenced RDF datasets. It serves as a role model for the SemanGit project, as they successfully extract and transform large scale Wikipedia data to a semantic dataset.

By going through the steps of creating a semantic version of the `git` data we draw from GHTorrent, we want to utilise the strengths of both RDF and graphs in general. More precisely, graph databases are good for structure exploring queries, as discovered by *Vicknair et al.* [12]. A classic example is the "friend of friends" example from social networks, often used for making friend suggestions. In a relational model, this requires expensive table join operations, whereas this can be performed efficiently in a graph model. In our scenario of open source software repositories, the corresponding task would be to suggest similar projects to a user, as we will discuss in Section 3.3. When trying to add further data sources to our dataset, the strengths of RDF will be very useful, as it allows

us to easily deal with heterogeneous data. Furthermore, interlinkage with other datasets promises further analytical options. From the GHTorrent dataset, we know the state and city of many users. With a DBpedia interlinkage, we could, for example, analyse if users from densely populated areas behave differently to others, or if users coming from a city with a university produce better code on average.

Queries on RDF graphs are usually written in SPARQL [13]. However, to query an RDF graph, one first needs to load the data into a triplestore. In our case, this already proves to be an issue due to the size of our dataset. With over 18 billion triples as of November 2018 and over 22 billion triples as of June 2019, SemanGit encounters classical big data issues. For comparison, DBpedia – the central dataset of the Linked Open Data (LOD) cloud [1] – has about 9.5 billion triples as of June 2019 [14]. Due to the time limit for this thesis and a delay in the provision of the infrastructure for our work, we had concerns if a full deployment can be achieved with these resources. It turned out that we only have a single commodity machine at our disposal [2], providing enough storage, but insufficient computational power for many analytical tasks on a large dataset, due to limited RAM, limited multi-core processing and slow storage.

The title of this thesis was formulated with much precaution, as we were not aware if querying the dataset would be feasible. Ideally, we would generate the data, load it into a triplestore and conduct several analyses, emphasising that the dataset is valuable for science and business. In the worst case, this thesis would have developed into a theoretical elaboration about use cases with an industrial deployment of the dataset, where more resources are available. In the end, we agreed upon three goals capturing as many outcomes as possible.

1. Capture information about `git` activity in a semantic dataset
2. Create a SPARQL endpoint on the dataset
3. Find use cases and evaluate query capabilities

These goals are independent of each other. In case that the deployment of the SPARQL endpoint would fail, we developed different strategies to obtain sufficiently relevant results for our third goal. The optimistic target of performing these analyses on the real dataset with SPARQL can be weakened to the usage of SPARQL queries on a smaller dataset,

[1] `https://lod-cloud.net/`
[2] Intel(R) Core(TM) i7-5820K CPU @ 6 x 3.30GHz , 64 GiB DDR3, 2x4TB HDD @ 7,200 RPM

either a representative sample or by restriction to a subset of relation types.

The approaches to achieve our goals are described within this thesis. Before we elaborate these in detail, we will give a motivating example to showcase the value of the dataset. In Section 3, we will discuss related works associated with these three goals, before we start with the technical details about the creation of the SemanGit data in Section 4. Afterwards, Section 5 discusses how this dataset can be used to obtain representative samples, without the need of deploying all data in a triplestore. This deployment is then described in Section 6, where we evaluate different triplestores and describe how we load our data into one of them.

2 (*) A Motivating Example: Trends in Programming Languages

In this section, we will present an example to motivate further analyses on our dataset. Due to a long wait time to load our dataset into a triplestore, see Section 6, we created this analysis as we waited for a queryable graph dataset. Therefore, queries in this section are SQL queries on relational tables. In Section 7.3.1, we will revisit this analysis with a graph database at hand.

Like natural languages, programming languages evolve over time. Maybe even more so, as constantly new programming languages are invented with some purpose in mind, such as being useful for statistical analysis or supporting new architectures. Seeing that SemanGit contains meta information about millions of projects [3] on GitHub [3], we have chosen to take a closer look at the languages used over time. In this section, we will analyse trends in languages used on GitHub, starting from October 2015 where the records about language usage begin in our input data from GHTorrent [5]. An analysis for the years 2012 to 2014 was performed by GitHut [15]. GitHub has also performed a language analysis, presenting the usage of the most popular programming languages over the years and which ones are gaining the most contributors [16]. We will strive to present interesting and more detailed insights on the occurrences of programming languages, complementing the analysis conducted by GitHub. This includes a discussion about suitable feature weighting on repositories to determine the popularity of languages.

2.1 (*) Preprocessing the Data

The raw data about language usage from GHTorrent cannot be used directly for this analysis. There are a number of issues that need to be addressed beforehand, such as time gaps between observations. The table describing the usage of programming languages by a project, `project_languages`, has the following columns:

```
project_id (INT), language (VARCHAR), bytes (INT), created_at (TIMESTAMP)
```

Each row of the table shows the number of bytes of a programming language, used by a certain project at a specific point in time. The issue with the data is the inconsistency in regards to update intervals. GHTorrent describes this table on their website as follows [17]:

[3] The January 2019 dump of GHTorrent contains 110,719,662 projects in the projects.csv file, some of which have been deleted on GitHub, but remain in our dataset.

Languages that are used in the repository along with byte counts for all files in those languages.

Multiple entries can exist per project. The created_at field is filled in with the latest timestamp the query for a specific project_id was done.

The table is filled in when the project has been first inserted on when an update round for all projects is made.

Unfortunately, the last paragraph means that we sometimes have updates in quick succession about one project's language usage, followed by long gaps without any information, sometimes multiple years long. Additionally, all byte values are complete values rather than incremental, displaying the size of the language in a project at that time, instead of showing how the size has changed due to an update.
We decided to even out the inconsistencies of the data in two steps:

1. Flatten the data: If multiple entries exist for one triple of (project, language, year), merge those entries, choosing average values for the byte size.
2. Fill gaps in data: If for a year, there is no entry for a project and language, choose the last known value instead.

This yields exactly one entry per project and language per year, if the language was used by the project during that time, otherwise no entry. We have performed preprocessing for the two time granularities *year* and *month*.

The steps described above with time granularity *year* can be performed with the following SQL commands:

```
--Step 1: Reduce to one entry per project and language per year
CREATE TABLE project_languages_average (project_id INT NOT NULL,
    langdate TINYINT NOT NULL, language VARCHAR(100) NOT NULL,
    langsize BIGINT NOT NULL) AS
SELECT project_id, DATE_FORMAT(created_at, '%y') AS langdate, language,
    AVG(bytes) AS langsize
FROM project_languages
GROUP BY project_id, langdate, language;
--Index is vital for efficiency in step 2
ALTER TABLE project_languages_average
ADD UNIQUE INDEX(project_id, language, langdate);
```

This results in an intermediate table that we will use for the next query, where we fill gaps in data with the last known value in case none is present for the year in question.

```
--Step 2: Fill gaps in data
CREATE TABLE proj_lang_annual_value (project_id int(11) NOT NULL,
    langyear TINYINT NOT NULL, language VARCHAR(50) NOT NULL,
    langsize BIGINT UNSIGNED NOT NULL);
INSERT INTO proj_lang_annual_value(project_id, langyear, language, langsize)
--Project ID, language and timestamp are already at hand
SELECT p.project_id, allyears.langdate AS langyear, p.language AS language,
    --Subquery to get the latest "size" value for the language
    (SELECT t.langsize
    FROM project_languages_average AS t
    WHERE t.project_id = p.project_id AND t.language = p.language
    --Choose the latest value before / of current year as value
    AND t.langdate <= allyears.langdate ORDER BY t.langdate DESC LIMIT 1)
    AS langsize
FROM
    --Join to get all projects with all their languages for every year
    (SELECT DISTINCT project_id, language, langdate
    FROM project_languages_average) AS p
    INNER JOIN
    (SELECT DISTINCT langdate
    FROM project_languages_average) AS allyears
    ON pdate <= allyears.langdate AND NOT EXISTS
        --Avoid duplicates by checking that no later value exists
        (SELECT 1 FROM project_languages_average
        WHERE project_id = p.project_id AND language = p.language
        AND langdate <= allyears.langdate AND langdate > pdate)
ORDER BY p.project_id, allyears.langdate;
```

The table creation of the first step took 6h43m plus another 35m for the index generation.

Step 2 took a total of 1h09m. Equivalent preprocessing with monthly grained values took an overall of 21h05m. The reduction step 1 with annual values only pruned 4.01% of the original data. This shows how very few full update rounds were performed by GHTorrent.

2.2 (*) Popularity of Programming Languages

At first, we will perform a very general analysis about how popular each language was in which year, starting in 2015, where the records of language usage from GHTorrent begin, to 2018. The computational effort of such an analysis is small, having performed the preprocessing described above. The main problem here is to choose a suitable score function for the popularity of a language. In GitHub's language analysis, the number of users collaborating on a project was used as weights for projects [16], which we will also address in Section 7.3.1. An analysis of this kind would not be well suited at this time where we work with the relational scheme only, as this would require expensive joining of multiple large tables.

We do have information about the byte size of languages for each public project stored on GitHub. This data seems to be a poor popularity indicator by itself though, as Table 1 shows. A much more realistic indicator is the number of occurrences of a language, meaning the number of repositories in which the language is used. The resulting top ten languages are shown in Table 2. A third strategy we have adopted is to sum up the relative sizes of the languages per project. A project consisting equally of JavaScript and HTML for example, would count 0.5 to both languages. The results are shown in Table 3. Finally, Table 4 shows the number of projects for which a given language is the largest in size. Note that we did not exclude any languages from the results which might not be considered to be a programming language. To get accurate values for Tables 3 and 4 which exclude non-programming languages, we would have to decide for nearly 400 languages whether or not it is a programming language.

If we discard Table 1 as irrelevant, the remaining tables share one drawback. None of them takes the structures of the related repositories into account. One could, for example, add weights to repositories according to the number of commits. However, this would favour repositories with contributors who do frequent and small commits over those with contributors who summarise their work in fewer commits.

Another possible strategy is the one conducted in the analysis of GitHub [16], adding weights to repositories based on the number of unique collaborators. In Section 9.2 we will discuss potential drawbacks of this strategy by introducing the concept of *ghost contributors* – collaborators of a project, who do not submit code to it.

Table 1: Languages by size

Language	Total Byte Size
etag	482,188,416,547,331
c	20,364,579,839,133
javascript	6,648,424,319,875
c++	5,194,731,644,235
java	4,061,350,050,534
html	3,476,905,493,554
php	2,773,664,743,557
python	2,121,105,092,157
c#	1,431,575,065,852
jupyter notebook	1,240,532,358,071

Languages and the sum of sizes in bytes across public repositories on GitHub in 2018.

Table 2: Languages by occurrence

Language	Occurrences
javascript	11,951,722
html	10,324,429
css	9,495,367
shell	5,904,000
java	5,745,568
python	5,320,801
etag	3,779,556
ruby	3,612,398
php	2,803,936
c	2,797,153

Languages and the number of their occurrences in public repositories on GitHub in 2018.

Table 3: Languages by relative size

Language	Sum of relative sizes
javascript	5,903,203
java	4,470,494
python	3,061,516
html	2,907,523
ruby	1,799,021
css	1,793,554
php	1,690,558
etag	1,684,910
c++	1,298,673
c#	1,158,371

Languages and the sum of their relative size across public repositories on GitHub in 2018.

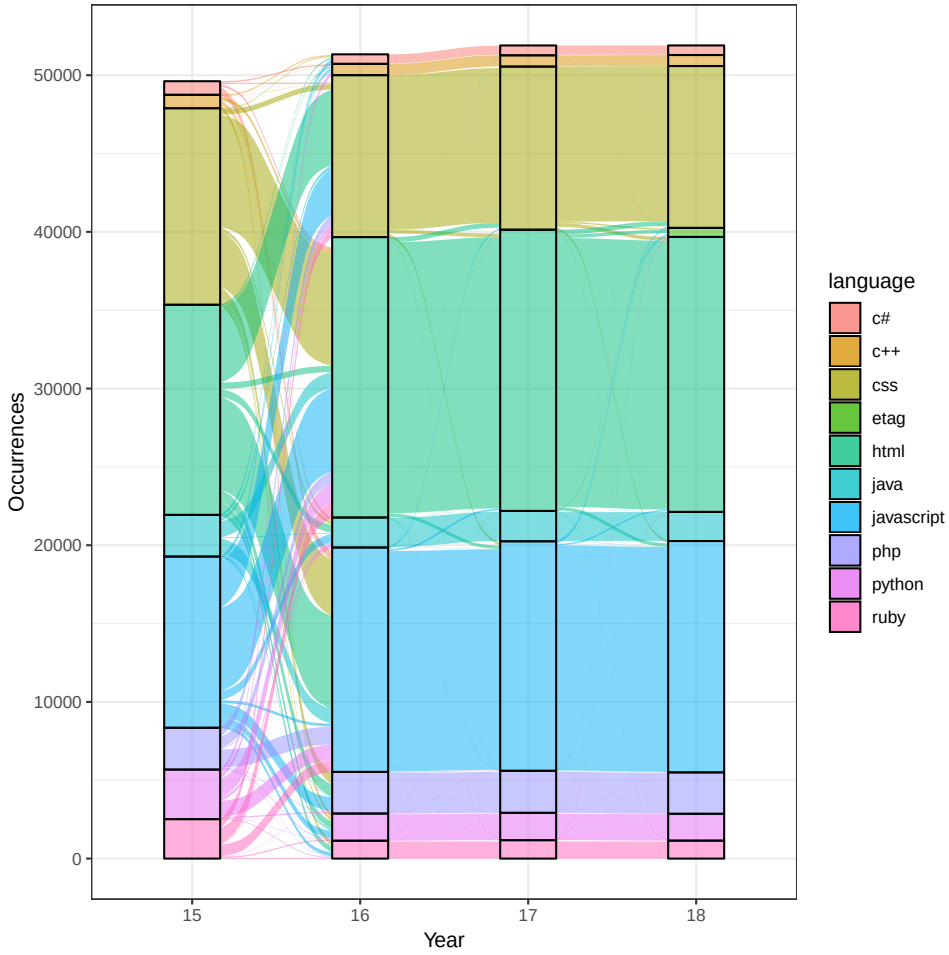
Table 4: Languages by primary occurrence

Language	Occurr. as primary
javascript	6,254,461
java	4,566,125
python	3,123,833
html	2,495,372
etag	2,467,028
ruby	1,936,145
php	1,812,335
css	1,447,463
c++	1,386,988
c#	1,188,564

Languages and the number of projects on GitHub in 2018 in which the language is the primary language, i.e. the largest.

Note that Tables 3 and 4 are very similar. They actually feature the same languages, just in a slightly different order. Their top three are also identical to the top three of GitHub's analysis [16]. The rest of the tables are more difficult to compare, as their language analysis is restricted to programming languages only, while we also have *HTML, CSS* and *ETag* in our results. From Table 2, one can guess that *JavaScript, HTML* and *CSS* have many projects in common, as they obtain similar scores on Table 2, but not on the other ones. It appears that many projects include a small amount of *shell* code, such as installation programs, causing *shell* to only appear on this table.

We will now use the results above to narrow down further analysis to relevant languages. Firstly, we present an alluvial diagram which shows how the popularity of the languages has evolved over the years, using the weighting function shown in Table 4,

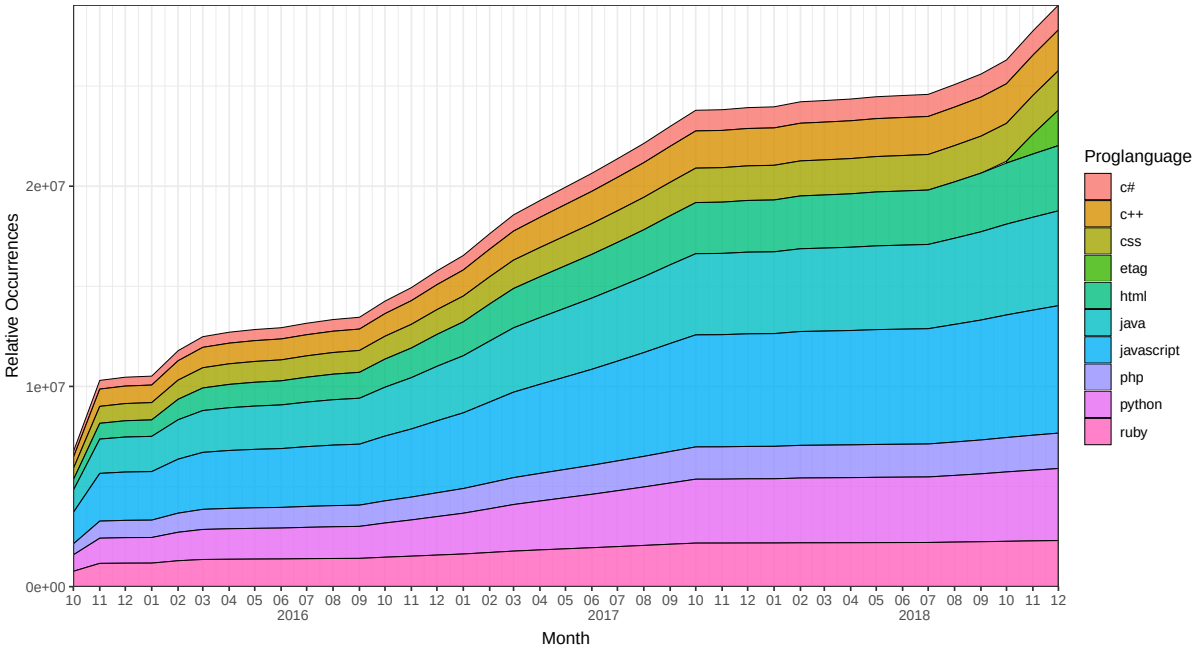


The alluvial chart visualising the occurrences for 10 programming languages for the years 2015 to 2018. For each year, the number of occurrences is represented by the height of the respective bar. In between the bars, the proportion of changes from one language to another is visualised. Each colour is assigned to a fixed language.

Figure 1: Changes in primary language per year

counting the repositories having a certain language as primary language. This analysis aims to not only give a yearly value indicating a languages' popularity, but also to explore how developers migrate from one favoured language to another. Secondly, we will attempt to give finer grained insights with monthly values, using the weighting function as described for Table 3, summing up the relative occurrences of a language.

To highlight the effect of the alluvial diagram, showing how projects migrate from one primary language to another, we have chosen to only include projects in the diagram which have changed their primary language at least once since 2015. This also demonstrates the poor update quality of the input data - many updates were performed during 2015 and 2016. In contrast, data from later years mostly contains information about new projects, instead of updates for old projects. Therefore, the diagram shown in Figure 1 is almost stagnant for later years.



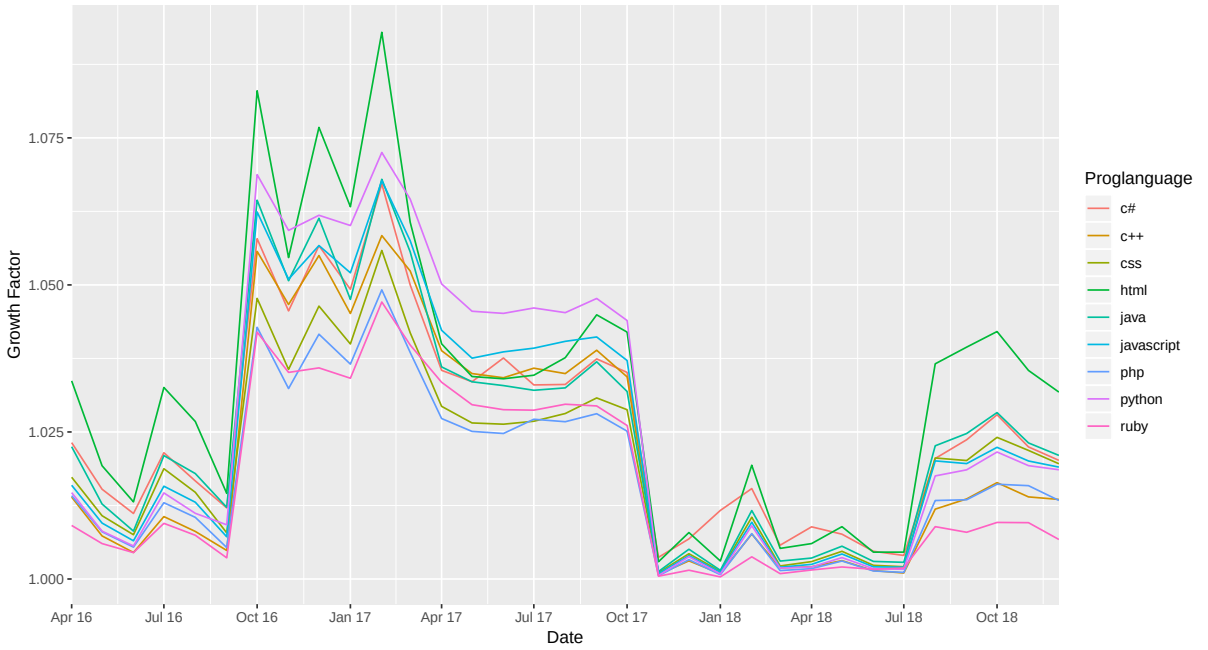
Stacked chart for the number of relative occurrences of programming languages. This number is represented by the height of the respective colour for a fixed month.

Figure 2: Relative occurences of languages per month

From the figure one can easily see the strong link between JavaScript, HTML, CSS and PHP, whereas other connections are more subtle. For example, Python seems to have lost many GitHub projects to JavaScript from 2015 to 2016 and gained only a few in return. The data is almost constant from year 2016 onward, which is due to lacking updates on the dataset. For this reason, it seems prudent to perform analyses which do not rely on full updates, but on new occurrences or an event based structure instead.

Next, we will look at occurrences of languages from a different perspective and drop our project based focus, taking more data into account. To do so, we created a stacked area chart, showing the general evolution of the top ten programming languages in Figure 2. The language *ETag* does not occur until October 2018. This is most probably due to GitHub not recognising *ETag* as a language prior to this date. From this diagram, it is difficult to see for any particular language whether it has grown or shrunk. The first general growth peak from October 2015 to November 2015 is due to the initial bulk imports not finishing before the end of the month. Other changes in growth rate, such as in October 2017, are more difficult to explain. The general reduction in growth could be seasonal, or potentially due to lost events, caused by software bugs in the data collection.

In further investigations, we computed a graph showing the growth rate for each lan-



Monthly growth of 10 most popular programming languages. The y-axis represents the growth factor, which is computed as quotient of two consecutive months.

Figure 3: Growth factor of languages

guage explicitly, see Figure 3. To make the result legible, some initial values and all *ETag* entries were removed from the dataset, as the corresponding growth factors otherwise overshadow all other values. Note that no growth factor is below 1. This is most probably due to the fact that existing projects are hardly ever updated in our dataset, see above, and we therefore only obtain growth from new projects. The strong increase in *HTML* as language could be due to the page hosting service of GitHub [18]. For some months, lots of data is present, whereas for others there is almost no data at all, being another example for the poor update quality.

The results show that a timeline analysis is not very suitable, given the low update quality of GHTorrent regarding project languages. We will therefore get back to analysing programming languages from a different point of view in Section 7.3.1, focusing more on geographical trends regarding language usage. This kind of analysis is expensive in a relational model though, as we need to perform many join operations, but it is quickly computable on a graphical model.

3 (†) Related Work

In the introduction, we described the research goals of this thesis. They can be summarised as follows:

1. Capture information about `git` activity in a semantic dataset
2. Create a `SPARQL` endpoint on the dataset
3. Find use cases and evaluate query capabilities

While the three goals are related to each other, they all reveal their own challenges and problems. Therefore, we have chosen to split this section into three parts, one for each research goal. Subsection 3.1 compares our approach with other attempts to mine information about `git` or the creation of large scale semantic datasets. Subsection 3.2 targets the big data issues that arise when loading the data into a triplestore and discusses research done on deploying a `SPARQL` endpoint as well as alternative tools that have been developed. Lastly, Subsection 3.3 shows the vast amount of analytical work that has already been performed on `git`, GitHub and other software repository providers, showing which analyses were already conducted on similar data.

3.1 (†) Creation of a Semantic Dataset

To create a large scale semantic dataset about `git`, we need to select a data source. As GitHub is hosting over 100 million projects as of April 2019 [3], making it the largest `git` provider on the web [2], and offers a `REST API` to query data about users and public repositories, we chose to focus on extracting data from this provider. The downside of using their API is the rate limit of 5,000 queries per hour for any authenticated user [19]. This implies that querying the projects directly via the API is infeasible for us with the time restriction of six months for our thesis, allowing for only 21.6 million queries in that period. We found that similar restrictions apply to other providers' APIs as well, such as the `GitLab API` [20] which allows up to 10 requests per second per client IP address.

GHTorrent [5] is a project that has been tackling this issue by mining the `GitHub API` with hundreds of community-donated authentication tokens, raising the number of queries they can perform per hour. Data dumps can be downloaded from their website [4]. This offers an alternative data source for our research, allowing us to deploy data on a local `SQL` server. One of the major drawbacks of using the GHTorrent dataset is that

[4] `http://ghtorrent.org/downloads.html`

their data is by no means complete. Even with multiple authentication tokens at hand, bulk updates of projects seem to be too expensive, as we noticed in Section 2, where the update frequency of the language usage of a project seemed to be too sparse.

There are several other issues that can arise about the data. As depicted in a study by *Kalliamvakou et al.* [21], there are some practical pitfalls that need to be avoided when working with data from GitHub. We will list the ones that have the greatest potential impact on our research:

- 87% of projects showed no activity within the last month of their study.
- "repositories" and "projects" are not the same. If a project is forked, a new repository is created. But this usually is not a new project.
- 71.6% of the projects have no collaborators besides the owner.
- Only a small amount of projects use pull requests.
- Pull requests only show as merged if merging is done via the GitHub website.
- About half the users do not work on public projects.
- Not all contributions are made by registered users.

GHTorrent already takes care of the last point, creating "fake users" if an author is not registered [17]. The other points need to be taken into consideration when performing analyses on the dataset. GitHub is not the only perilous `git` data source though: *Howison et al.* [22] also show that SourceForge data comes with similar issues.

Besides GHTorrent, another notable database is Boa [23]. It is an infrastructure and language for mining software repositories at large scale, which can be accessed via a web-based interface. One key advantage is the easy re-usability of research done with Boa, as experiments can be repeated by simply re-running a query. Despite this, Boa is not widely used by researchers investigating GitHub: In 2017, *Cosentino et al.* [4] summarised the methods, datasets and limitations on research papers concerning GitHub. 41.25% of the papers used GHTorrent as data source, another 31.25% are querying the GitHub API directly. Boa is part of the category "others", which were used by only 3%.

Despite our initial focus on GitHub, our infrastructure is designed to handle data from several sources, see Section 4.1. Without sufficient computational resources though, we are not able to integrate further data sources into our GitHub dataset. However, future work might include the integration of SourceForge data into SemanGit. In such a case, it is useful to find out which developer accounts across these platforms belong to

the same person. How this can be achieved has been investigated by *Robles et al.* [24]. Unfortunately, their techniques cannot be applied to our case, as they use heuristics on real life names and email addresses, as well as usernames. With the European General Data Protection Regulation (GDPR) in place, GHTorrent was forced to stop distributing such sensitive information. While some names and addresses can still be found in old GHTorrent data dumps, using this data would raise severe privacy concerns.

While we did not manage to load further external datasets into our triplestore, we managed to interlink parts of our data with DBpedia [11]. GitHub users have the ability to add a location to their profile. This location is geocoded by GHTorrent, yielding the city and state that correspond to the location, if applicable. One needs to keep in mind that users can enter wrong information, or even imaginary places. If a state and/or city can be determined, we refer to the DBpedia resource, e.g. the city *San Francisco* will yield `<http://dbpedia.org/resource/San_Francisco>`. This provides a multitude of new analytical options that cannot be easily achieved without an interlinked graph database, as additional information for user locations, such as the capital, latitude and longitude, population, nearest city, time zone, populated area, population density or the mayor of the city can be taken into consideration. The strength of RDF dataset interlinkage is demonstrated by the growing size of the Linked Open Data cloud [25], which contains 1,239 datasets with 16,147 links as of March 2019 [5]. Looking back five years to August 2014, the diagram only contained 570 datasets.

[5] https://lod-cloud.net/

3.2 (†) Adding a SPARQL Endpoint

In general, the deployment of a SPARQL endpoint is not a difficult task. It boils down to the installation of a triplestore and filling it with data. Most modern triplestores offer an automatic deployment of the server infrastructure together with APIs. The `W3C` collected multiple triplestores, together with the reported number of supported triples [26]. All of the 18 listed solutions provide a SPARQL endpoint. Concerning the size of our dataset, 7 out of them seem to be appropriate for sizes above 20 billion triples. Unfortunately, in many cases these reported numbers were obtained on clusters of at least 3 machines or on servers with superior specifications. The highest reported value so far is `Oracle Spatial and Graph with Oracle Database 12c` with 1.08 trillion loaded triples on a setup consisting of 8 nodes with 24 cores each and overall 2TB of RAM [27]. For comparison, the smallest listed study for `Oracle Spatial and Graph` listed, claims to load around 1.1 billion triples in just 28 minutes with specifications expected to be highly superior to ours, consisting of a rack with 16 quad core processors, 512GB of RAM and 160 flash drives [26]. Studies for `Anzograph` using 200 nodes, having 16 cores and 208GB of memory each [28], `AllegroGraph` with a rack of 240 cores, 1.28TB memory and 88TB hard disk [29], as well as `OpenLink Virtuso` with 8 nodes summing up to 256 cores and 2,048 TB of RAM, all suggest that it is possible to load more than 20 billion triples in acceptable time, but with unrealistically expensive infrastructure.

As we expect to have only one server with off-the-rack specifications, we need to focus on solutions appropriate for a single server deployment. *Corcoglioniti et al.* claim that there is no adequate tool for processing large scale RDF datasets on a commodity machine [30]:

> *Tools scaling to large datasets typically employ parallel, distributed computation models such as MapReduce(e.g., [14, 24, 25, 1]) and cannot be used efficiently —if not at all— on a single commodity machine. Tools targeting local computation, on the other hand, are often based on some form of data indexing (e.g., [19, 12, 24]), such as a triple store or a similar index [16]; these approaches typically scale as long as the index can be kept in main memory and incur a severe performance hit when data has to be accessed on disk.*

The authors develop their own tool, capable of processing large scale `RDF` datasets on a single machine. Unlike conventional triplestores, their tool suite is not targeted to implement a persistent storage, but to collect statistics and perform processing tasks on

the RDF dataset by using streaming techniques. The supported operations include the extraction of the TBox, filtering, normalising and mapping between vocabularies, inference materialisation and the creation of owl:sameAs relations.

We therefore conclude that the deployment of a SPARQL endpoint for this size of data is challenging and that we might have to use additional technologies to reduce our data size. One such approach is described by *Manolescu et al.* [31]: Instead of deploying a SPARQL endpoint on the original dataset, they suggest the generation of a summarised graph of considerably smaller size, compressing several facts automatically. While the authors provide a tool suite for executing the compression, the run-time analysis states a time factor of $O(|G|^2)$ for generating the summarised graph and their architecture employs a triplestore in the backend.

Another possibility is the sampling of RDF data. *Rietveld et al.* [32] conclude that most realistic queries only require 2% of the stored data for the example of 5 large scale RDF datasets. They discuss the idea to learn weights on subsets of the dataset to implement importance sampling. The authors also provide a Hadoop and Python backed tool for generating these samples. While their reported run-times seem to be reasonable, the usage of their methodology requires the evaluation of tuples of training queries and results on the full dataset.

The idea of sampling representative subgraphs is not well discussed in the case of RDF, but there are contributions concerning general graph sampling. *Hübler et al.* [33] discuss this problem from a mathematical perspective. They show that deciding if a sample is representative is NP-complete and propose a probabilistic sampling method, based on the Metropolis Algorithm [34]. While some operations, like randomly selecting a node, can be directly processed in the *SemanGit* RDF text files, the iteration over adjacent nodes would again require the possibility to run graph traversals efficiently, i.e. having a triplestore. Alternative approaches lead to the same problems. *Lu and Bressan* [35] evaluate three different methods for subgraph sampling, utilising techniques like Markov Chain Monte Carlo, Random Walks and Neighbourhood Sampling. They all require the ability to query neighbours of a selected node.

As the previously mentioned alternative approaches all require to load the full RDF graph, we turn our research to stream-based solutions like the already mentioned tool

of *Corcoglioniti et al.* [30]. Indeed, the idea of streamed analysing RDF data is discussed several times within the area of RDF Stream Processing (RSP) and a wide range of tools and extensions to the SPARQL query language have been developed. Again, a listing of these tools can be found at W3C [36]. While the implementation of a SPARQL endpoint and repeatedly streaming the SemanGit RDF would be too imprecise for most analyses, this technology can be used for generating graph samples. This idea is already discussed and implemented by *Dia et al.* [37], who implement an extension to the C-SPARQL language, enabling three different ways to sample subgraphs out of a stream of RDF data. Unfortunately, the authors do not provide any reference to the source code and we could not find any reference in the C-SPARQL research group's GitHub [38] repository.

In conclusion, most tools discussed above cannot be used without severe performance issues on our setup. We have to expect slow or limited deployment of our dataset in a triplestore. In Section 5, we discuss sampling methods based on approaches not using a triplestore or trying to reduce to only the necessary functions. Afterwards, we try to evaluate the possibilities of deploying a SPARQL endpoint by ourselves in Section 6.

3.3 (*) Use Cases

The approach to use and query data from GitHub is not a novelty [4] and especially social scientists showed their interest in such data. Even before the invention of the `git` protocol, prominent economists like *Tirole et al.* tried to elaborate the motivations behind open source software development [39]. Their work consists at the point of its publication of statements and hypotheses, derived from prominent examples, which are not verified by any field study. With the `SemanGit` dataset, we have the means to analyse the personal behaviour of users and organisations at large scale, all voluntarily publishing their open source codes online.

We will discuss use cases of data extracted from GitHub with references to related work done in this field, moving from purely social interaction based analyses to other applications, such as personal behaviour, usage and distribution of programming languages or geographic data.

As already described above, there are certain limitations to the dataset used. Analyses on GitHub data need to be evaluated carefully, as they are prone to miss-interpretation [21]. examples are that not all repositories are projects, most projects are inactive, many users do not contribute to public repositories, and pull requests only show as *merged*, if they were merged via the GitHub website, see Section 3.1. Not all GitHub repositories can be grasped by analyses, as users have the option to create private repositories, making them inaccessible for unauthorised users. This also limits data about repository related interaction, such as creating a commit or watching the private repository.

A user's profile is always public and filled with automatically generated information related to two distinct social features: A user can follow another user or watch a project. These mechanisms are derived from social media platforms, which are target of social analyses since they gained popularity in the 2000's [40–43]. Obviously such research can also be conducted on GitHub. An early contribution in this direction is done by *Thung et al.* [44], published in 2013. They extracted 100,000 projects and 30,000 developers from GitHub and perform a social analysis on the resulting graph. To determine who or what is influential, they apply a `PageRank` algorithm, based on the set of common developers of projects and users following each other. The ability to watch entire projects was not taken into consideration, possibly because this feature was revamped shortly before in August 2012 [45]. Focusing also on the ability of watching projects, *Blincoe et al.* target

the related question, how popular users influence others. The small study with 800 users analyses followers who star, contribute to, or fork a project after a popular user committed to it. Starring a project is similar to watching it, except that no update notifications will be sent. In contrast to other studies, they combine the online gathered data with data from questionnaires, to capture the motivations behind those actions [46].

Moving to a larger scale of data, *Yu et al.* [47] query the follow relations of 1.5 million users, create a graph of all users having at least 5 followers and cluster for popular subgraph patterns, like isolated groups. While such an analysis can be conducted at a single point in time by querying the current state, *Jiang et al.* extend the approach on technological base [44]. They analyse the behaviour of unfollowing other users, by collecting such events from the GHTorrent [5] daily dumps over a period of 5 months, and enriching the data with `GitHub API` calls to contain 700,000 users as statistical population. A similar attempt of conducting questionnaires like above [46], seems to be not as significant due to lacking participation of users, with less than 30 answers to some questions. Not only the structures that followed the implementation of social features, but also the benefit of these social features itself is a topic of discussion. *Bogdan Vasilescu* elaborates the effects of the social media components on programmers behaviour [48] for GitHub and other Platforms. The author collects publications concerning different aspects of social mechanisms and their effects on users' interactions. Although he also discusses how these effects might affect analysis of the data, the entire work is a theoretical elaboration, missing any real-life examples. A concrete approach based on data from GitHub was formalised by *Low et al.* analysing how the usage of social media features might improve the maintainability and long-term support for projects [49]. They conclude by analysing the extracted patterns, that the survival of a software project is mostly dependant on the spreading of awareness along the social network. Not only from the perspective of project maintainers, but also from a user perspective this social technology can be used to achieve a better development. *Zhang et al.* for example propose different patterns of user behaviour to recommend software projects to others [50]. These features are then extracted from a dataset containing the 86 most popular projects of GitHub and analysed in a Latent Feature model.

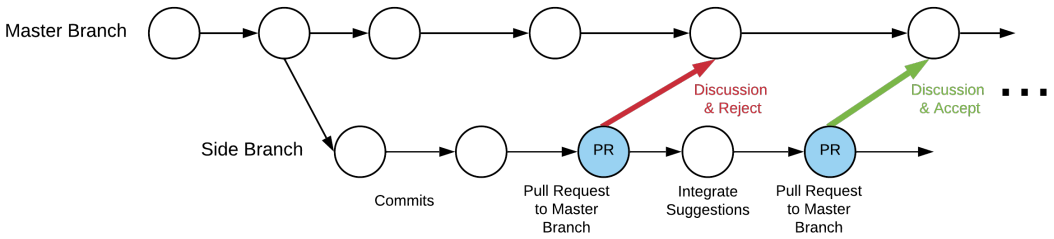
Despite the social structure, SemanGit also contains additional information: Issues, project structures and commit behaviour, that might be employed for an in depth analysis of the actual performance of collaborators. A use case that might be relevant for several

business areas, is the acquisition of new employees. *Sachdeva* [51] evaluates multiple algorithms for automated recruiting of programmers, that extract keywords from job postings and matches users from the `GitHub API`. While we are not aware of a working system that was published, the author provides a set of queries for the `GitHub API` to obtain recommended programmers.

If a developer has a lot of followers on GitHub, this can be interpreted as an indicator that he or she makes good code, contributes on a regular basis, is active in popular projects or uses common programming languages. *Filho et al.* investigate how influential Brazilian developers on GitHub, i.e. developers with many followers, are distributed across the country [52]. They found 4,000 GitHub users who state on their profile that they live in a place somewhere in Brazil. The geographical distribution is cross-referenced with socio-economic data about the region, such as the number of educational institutions in a state or its gross domestic product. An analysis of this type could also be performed with the SemanGit dataset, especially since it is interlinked with DBpedia and includes geocoded information about many developers. We also demonstrate how a follower-based analysis can be performed with our dataset in Section 7.2.

Another key feature of `git` is *branching*, as it helps developers organising their distributed collaboration process. Instead of chaining all commits into linear order, projects can be split up into several branches, where one branch is marked as the "master branch". For example, if a user is working on a new feature for a program, she could create a new branch and commit all feature related contributions to it. Once finished, the newly formed branch, now containing a couple of commits, can be merged back into the original branch. This step is often done via a *pull request*, which provides features for code reviewing and discussion. A contributor of the project needs to accept the changes before they are merged. If users wish to contribute to an external project where they have no permission to push commits, they can *fork* the project. This creates a copy of the project to which they can push commits, and also send their contributions back to the original project via a pull request. An illustration of this process is shown in Figure 4. In SemanGit, projects which have been forked have a "forked_from" relation to the original project. Note that pull requests and forking are provider-specific features and do not belong to the `git` protocol.

As branching helps to organise distributed collaboration within a project, analysing



The illustration shows the mechanisms of branching and pull requests. Each node represents a single commit. The upper lane represents the master branch, the lower lane a side branch. Every interaction between those branches is marked by an arrow in between the lanes. After the second commit to the master branch, a user decides to create a new commit in a separate branch. The development in both branches continues independently of each other for several further commits. One user decides to start a pull request from the side branch to the master branch, which is rejected with feedback for improvements. After the integration of the suggestions, a second pull request is started and accepted, which leads to a merge of commits from both branches. Both branches continue to exist.

Figure 4: Intra-project branching and pull requests

the branching structure of repositories can present good insights. *Lee et al.* [53] report that analysing on branch-level is better suited when analysing collaborative work of developers than working on commit-level, as this better highlights the decentralised structure of the project. While `SemanGit` contains pull request data, it does not contain branching information yet. GHTorrent excludes some relations, such as branching, from the monthly dumps, and only provides them in the incremental daily dumps. As the deployment of the `SemanGit` dataset extracted from the monthly dumps is already challenging with our resources, we have chosen to keep using the monthly dumps until further resources are available. For comparison, the June 2019 monthly dump, which is a full database dump – except for the missing relations – has 103GB, while a single daily dump from 2019, containing one days' updates, usually has a size of around 5GB to 8GB.

When analysing the competence of a GitHub user, it is important to keep in mind that other factors than code quality can affect the results. There are indicators that not all users judge the code contributions of an external pull request purely on quality. Indeed, *Rastogi et al.* [54] find that the acceptance ratio of external pull requests is correlated to the origin of the author and reviewer: They conclude that for pairs of countries which are politically involved, such as Germany and China, the acceptance of pull requests from one country to the other is significantly lower than for uninvolved pairs, such as Japan and Switzerland. This can cause side effects that need to be considered when performing pull request analyses. We also perform a geographical analysis on global cooperation in Section 7.1, investigating which countries frequently work together in projects.

Besides politically motivated decision making on pull request acceptance, *Terrell et al.* [55] have also discovered a gender-based bias: While women tend to have higher acceptance rates on pull requests overall as compared to men, they still suffer lower acceptance rates, if their gender is clearly identifiable and if they are committing to external projects. The authors conclude that while female developers on GitHub may be more competent, they still suffer from a gender bias.

Takhteyev et al. [56] also performed a GitHub analysis with a geographical focus. They investigate different regions, comparing the number of GitHub users with that location, the share of projects and various other characteristics of these projects, such as the number of received contributions or watch events. From this, they find that North America receives an unproportional amount of attention, as the share of watch events on North American projects is larger than the share of North American users. This article was published in 2010 where GitHub only had 1 million projects - around 1% of the data we have at hand now.

Rusk et al. [57] perform a different type of geographical analysis, taking the usage of programming languages into account. Similar to our analysis in Section 7.3.1, regional differences in language usage are compared. In their work, they aggregate the usage of programming languages for a location and then generate a list of developers within the location that use this programming language. Our analysis does not break down to individual levels, but rather compares the differences in language utilisation across nations.

Investigating the usage and evolution of programming languages is a classical analysis, as also performed on `GitHut` [6] and GitHub [16]. A recent work of *Celińska* [58] focuses on how users can gain coworkers for their projects. Her results state that reciprocity is a factor in the open source software community – helping others increases the chances of them helping in return. She also find that influential people tend to find more collaborators, and that the coworker recruitment success is correlated to the programming languages used.

While several of the works mentioned above study influential users or projects, the influence was usually measured by a user's number of followers or a project's number of watchers. Another metric is introduced and discussed by *Badashian et al.* [59], which

[6] `http://www.githut.info`

measures how often a project is forked. Using these three measures of influence, they provide a list of the top ten programming languages on GitHub, as compared to our analysis in Section 2. Furthermore, they show that different types of entities (engineers, CEOs, authors, professors, speakers, organisations) attract different types of social interactions (following, watching or forking projects). We discuss how the role of a person within an organisation might be discovered, based on such interactions in Section 7.2.

Apart from trying to find behavioural traits by analysing developers' social activities on GitHub, data that is not socially related can also be utilised. *Coelho et al.* [60] analyse the issues that developers posted on GitHub. By collecting and extracting almost 160,000 reported issues related to Android projects, the authors manage to detect new, undocumented error and bug sources in the Android and proprietary libraries.

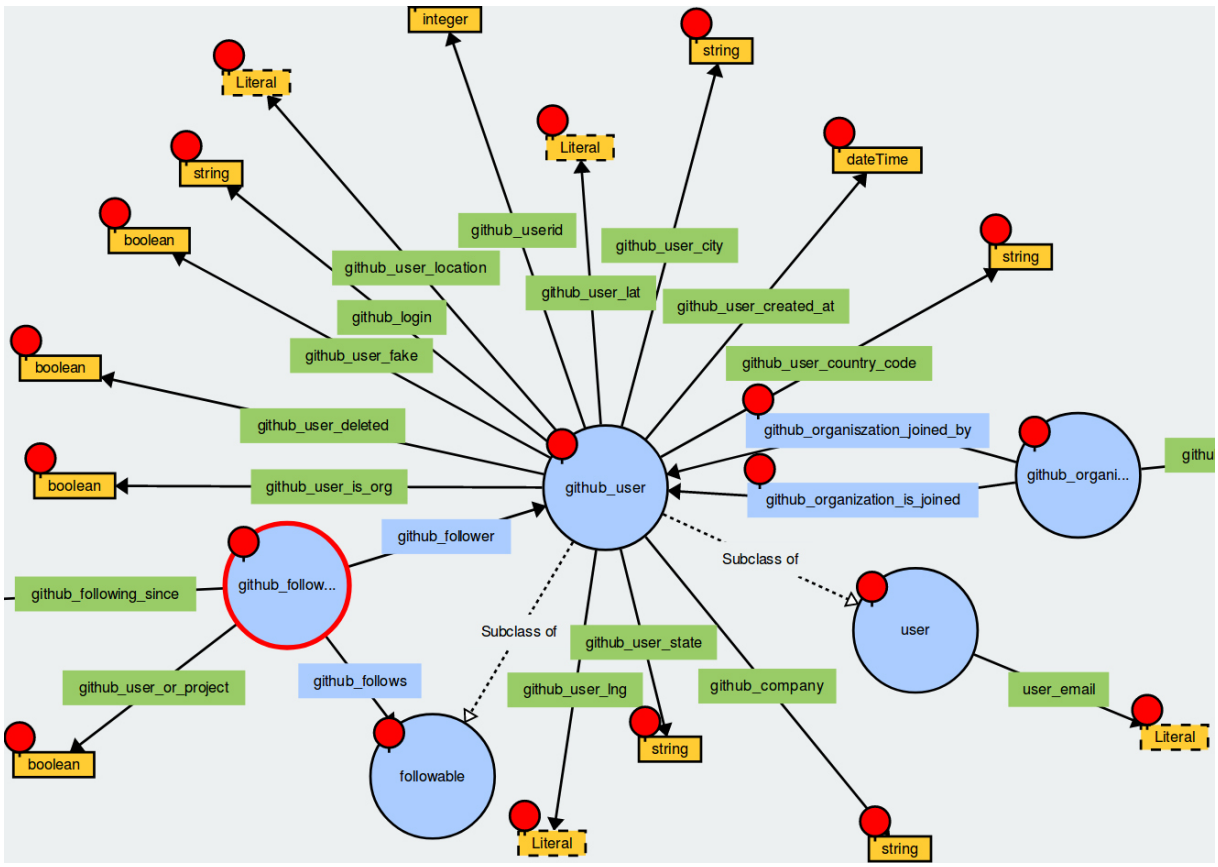
4 (*) Creation of the SemanGit Dataset

In this section, we describe the steps taken to obtain the SemanGit knowledge graph. Prior to this thesis, we designed an ontology that we briefly summarise in Subsection 4.1. Similarly, we also created a conversion tool previous to this work, filling our ontology with data from GHTorrent [5], as shown in Subsection 4.2. Finally, we will elaborate how we improved the conversion tool and dataset since and how we made them publicly available. A more detailed description about the ontology creation can be found in our project report [61].

4.1 (*) Ontology

Prior to the creation of a knowledge graph about `git`, an ontology is needed to structure the data and to enable logical inference. Seeing that `git` in itself is merely a protocol, we to collected data from `git` repository providers. There are a number of such providers on the web, the largest one being GitHub [2], reaching a global Alexa Rank of 44 in May 2019 [62], whereas the second highest ranked source code host is SourceForge with a global Alexa Rank of 359 for the same time period [63]. For this reason, we chose to gather data about GitHub, but keeping our ontology extensible to other hosts. This was achieved by using a hierarchical approach for the classes in the ontology. We have classes which are general to `git` and provider specific subclasses that inherit from the `git`-related classes, allowing us to capture host specific custom data and information about additional features. As an example, a `git` user is just a pair of name and email address, whereas a GitHub user also has a location, language, creation date, company related information and more. The location that GitHub users enter on their profile is available to us as geocoded data, allowing for an easy interlinkage with DBpedia [11]. The GitHub user class and its fields are prefixed with the host's name to clearly distinguish between the `git` protocol and provider specific features. An illustration that was made with the WebVOWL Editor [6] is shown in Figure 5. The full ontology, without extensions to other providers than GitHub, consists of 22 classes and 80 properties and can be found on our GitHub repository [7]. We show how the ontology can be extended to include GitLab in Section 8.1. To give some examples, there are classes for users, projects, languages, commits and the GitHub specific issue tracking feature, as well as classes capturing events, such as one user following another, which is also a GitHub specific feature.

[7] `https://github.com/SemanGit/SemanGit/tree/master/Documentation/ontology`



Visualisation of an excerpt of the ontology with WebVOWL [6]. This excerpt shows parts of the ontology modelling the characteristics of a user on GitHub. Classes are represented as blue nodes, relations as green or blue labelled arrows and literals as yellow boxes. `user` belongs to the classes obtained from the `git`-protocol. `followable` belongs to the classes introduced for the modelling of social interaction. All other classes represent data that is specific to GitHub and are prefixed accordingly.

Figure 5: Excerpt of the SemanGit ontology

We are aware of some limitations regarding the ontology, which we cannot address now without invalidating all queries from subsequent sections. In a few places, existing ontologies should be re-used, for example `foaf:mbox` for a user's email address. Some inverse properties should be added where applicable and naming conventions should be adopted [64]. These will be discussed in detail in Section 10 and we will strive to address these issues in an upcoming version.

4.2 (*) Converter

With the ontology in place, we can start the creation process of the knowledge graph. Unfortunately, it is infeasible for us to query data from GitHub directly, due to the fact that their API has a limitation of 5,000 requests per user and per hour [19]. As of November 2018, GitHub is hosting over 100 million repositories [3]. With the strong underestimation of requiring just one query per repository to gather all relevant information,

this would already take over 830 days [8], if done with one authentication token. Realistically, more queries are required, for example to obtain the language usage information for a repository that we used in Section 2. We were therefore forced to use a different input source. The GHTorrent project [5] has been gathering GitHub metadata for over five years, using more than 400 community donated tokens to be able to perform more queries per hour. Our Java conversion tool therefore translates their relational data into RDF. We will summarise some key facts about the tool we have created and compression methods applied that helped us to drastically reduce the disk size of the resulting dataset.

The GHTorrent monthly dumps consist of a set of .csv formatted tables and either contain a subject along with its literals or relations between entities. Our converter parses these tables row by row and translates the data with the usage of multi-threaded procedures. The fact that all tables in the GHTorrent data dump are presorted by an identifying column helps us greatly with space efficiency: When using the Turtle format [65], we are able to abbreviate as many successive triples as possible, as for any relation, all occurrences of an entity within the sorted column are listed successively. Especially tables like the *project_commits*, storing which commits belong to which project, let us abbreviate all but the first triple for any project.

Furthermore, we have created one prefix of at most two characters for every class and relation in our ontology, allowing us to drastically shorten the URI lengths. One character prefixes were given to the most commonly used classes and relations, while using the empty prefix for the most commonly used relation: `github_repository`.

Another technique we employ to reduce the output size is to change how all integers within entity identifiers are represented. Instead of using the characters `0-9`, we are using `0-9a-zA-Z` and the underscore (`_`) as alphabet for integer representation. Adding more characters to the alphabet would require the usage of non-ASCII characters, which would worsen the storage size when using UTF-8 encoding. A leading minus character on URIs led to errors in RDF syntax checkers, so we left it out, reaching an alphabet length of 63. The code of the integer conversion function is shown below.

[8] $\frac{100{,}000{,}000\text{queries}}{5{,}000\text{queries/h}} = 20{,}000\text{ h} = 833.33\text{ days}$

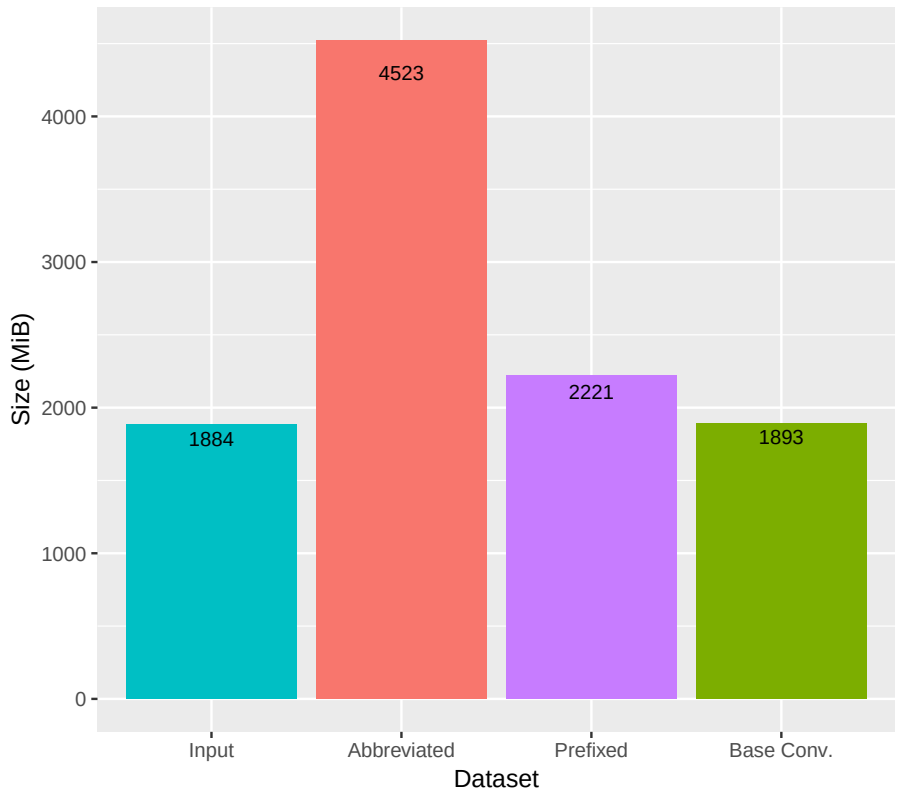
```
private static String integerConversion(String input){
  String alphabet =
    "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_";

  String rightOfColon = input.substring(input.indexOf(":") + 1); //numeric
  String leftOfColon = input.substring(0, input.indexOf(":") + 1); //prefix
  int in = Integer.parseInt(rightOfColon);

  //base conversion
  int j = (int) Math.ceil(Math.log(in) / Math.log(alphabet.length()));
  for (int i = 0; i < j; i++) {
      sb.append(alphabet.charAt(in % alphabet.length()));
      in /= alphabet.length();
  }
  return leftOfColon + sb.reverse().toString();
}
```

Combining these techniques, our resulting graph is only slightly larger than the input CSV formatted files. The November 2018 dump, which we initially used for the development, has a size of 304GiB. Our converted dataset requires 310GiB. An illustration of the output sizes when different methods are applied is shown in Figure 6, where we apply Turtle abbreviations, then add prefixes and finally also base conversions, i.e. changing integer representations of IDs. If we convert the output to N-Triples format [66], and don't use base conversion, meaning none of the compression methods are applied, the output size for the sample is 16,560MiB. Further statistics on the dataset will be discussed in Section 6.

4.3 (*) Improvements and Publication of the Dataset

Many of the points from this section up to here were done to some extent during a lab assignment in Summer 2018. A detailed project report is available online [61]. We have constantly improved the converter since, fixing errors in the RDF output, removing inconsistencies with the ontology or adding additional options for the output format, see Section 5. Some of the key changes will be discussed now.

We have experienced that the data we draw from GHTorrent has changed in structure multiple times. Columns were added to some tables to present new information, whereas



Plot of the space requirements for representing 1884 MiB of data from GHTorrent. The input is a CSV file. The other bars represent the size of RDF files. "Abbreviated" makes use of Turtle abbreviations. "Prefixed" uses prefixes of at most 2 characters plus abbreviations. "Base Conv." changes the integer representation to base 63, plus abbreviation and prefixing. The bar for N-Triple (16,560MiB) is omitted to improve the readability.

Figure 6: Output sizes for various compression methods

other tables ended up with fewer columns, as certain information had to be removed from their dataset due to changes to data protection law. The `repo_milestones` table, which should document the progress of projects towards self set goals, is completely empty now. This information is only distributed via incremental daily dumps, which are significantly more expensive to use, as the additional tables lead to a larger dataset. In the first version of our tool, developed in the lab, this would have caused our conversion process to fail by either crashing or creating wrong triples without warning. We have implemented new features, such as additional consistency checks using the `schema.sql` file, which contains the table structure information, and making sure that the structure is compatible with our current conversion process. Otherwise, as much as possible will be translated, while incompatible relations are skipped and warnings are produced. Once the conversion is completed, statistics about the percentage of incompatible lines are printed for every CSV file. High rejection rates indicate changes to the table structure. Furthermore, large portions of the converter were rewritten to modularise the code for better maintainability and to make computation of statistics on the dataset easier. Further desirable changes

will be discussed in Section 10.

Previously, we did not have the resources to load our dataset into a triplestore. This made it difficult to verify the integrity of the dataset beyond basic RDF syntax checking. During the process of creating a queryable knowledge graph, we have encountered many challenges due to the size of the data, which forced us to make adaptations to the output for performance reasons, see Section 6. In Section 8.2, we show how we interlinked our dataset with DBpedia. More specifically, the location information on user profiles is geocoded, allowing us to access additional information about the user's city and state from DBpedia.

Additionally, we have made the dataset publicly available on the web, including a documentation about the ontology and how the dataset can be reproduced using our tools [9]. We strive to update the uploaded dumps every 2 months, starting February 2019. With the most recent version of June 2019, the interlinkage to DBpedia is available.

[9] `http://semangit.de`

5 (†) The Challenge of Representative Sampling

During the early stages of this master thesis, we did not know what computational resources we would be able to work with. As described in the elaboration of our third goal in Section 1, we need an alternative approach for our analyses if the deployment of our SPARQL endpoint on the full dataset should fail. Although we found and listed literature talking about sampling of graphs in Section 3.2, we already summarised that most of these methods are not feasible for us. This is by no means a trivial task: The problem of deciding if a sample is representative, was shown to be NP-complete [33].

While there are promising sampling methods for graphs, their implementation poses technical difficulties, as neighbouring nodes need to be accessed frequently during graph traversals. Therefore, we either need to find a possibility to query for the neighbours of a vertex quickly, or we need to find an appropriate sampling method that does not rely on graph traversals.

In the following, we will discuss four different approaches, starting with the most trivial one of randomly rejecting triples outputted from the converter in Subsection 5.1. We use the results from the discussion, to develop an improved method, combining a greedy strategy with random rejection in Subsection 5.2. Thirdly, we will analyse the attempt to efficiently query neighbourhoods without a triplestore in Subsection 5.3, with the goal to draw connected subgraphs at random. Lastly, we will discuss the idea to generate representative datasets in Subsection 5.4, by utilising the sorted structure of our input tables, see Section 4.2.

Before we elaborate these four different approaches of sampling, we want to give a short reminder about the structure of input and output of our converter. The GHTorrent project provides us with data split up into several tables, provided in the form of CSV files. The converter transforms these tables into the RDF Turtle format, by translating row after row according to our ontology, abbreviating where possible. This process is parallelised to translate all tables at the same time, creating a separate RDF file each. These files are later merged to a single file by default.

5.1 (†) Totally Random Sampling

Our first idea is the naive approach of randomly choosing a set of triples from our knowledge graph. Formulated in a different way, we want to reject p_{reject} percent of the triples outputted from the converter. Of course, this method should not be applied to triples with relations associated to the vocabularies of `rdf:` and `rdfs:`. We exclude such constructs from the rejection process and only apply it to triples containing relations of the SemanGit: vocabulary. In the following, we will discuss some advantages and disadvantages of such a sampling.

When analysing the performance of such a sampling, two arguments are to be considered. On the one hand, fewer write operations occur, but each triple requires the effort to draw a pseudo random number. In the end, such a sampling procedure should require as much time as the converter itself, making it the first sampling method to run in feasible time.

Unfortunately, the short runtime is the only advantage of this random sampling. The drawbacks of such an implementation lie in the structure of the sample. As all rejections were drawn randomly, the loss of information would be evenly distributed across all nodes in the graph. For example, some users might only have literals for their location, some others only information for their email. The result would be a sparse graph, with a lot of contained entities, but a small portion of relations that appear per entity.

In the end, such a sample would only provide little value for most analyses. We therefore need a method of sampling which produces large sparse graphs within a similarly short runtime.

5.2 (†) Random Sampling with Greedy Collection

The main problem with the method previously mentioned is that the resulting graph is too sparse, due to the independent rejections. One approach to target this issue is to accept all triples containing the same subject together. The computational effort that is involved in collecting all these triples depends on the original format of the data source. Actually, we already used a weaker form of this approach. In Section 4.2 we elaborated that the structure of the GHTorrent dataset is good for abbreviations: All tables are sorted by their first column and all literal values of an entity are stored in the same row

within the CSV files. By applying sampling to each abbreviated block as a whole, we could greedily pick as many connections as possible.

We implemented this sampling method, as it only required minor changes to our current code. After generating as many consecutive triples as can be abbreviated, we can reject this set of triples with probability p_{reject}. The computational effort should be even smaller to the method described in 5.1, as we need to generate fewer pseudo random numbers and the abbreviation ratio of the sample is similar to that of the full dataset. A test of the runtime nevertheless showed a small time overhead compared to the conversion of the full dataset.

This method should definitely solve our problem of sparsity, but still has multiple disadvantages. One major drawback of rejecting and accepting entire sets of triples is that the output size can vary greatly, as not all resources have the same number of triples associated with them. For example, some projects have hundreds of thousands of commits, all abbreviated into a single statement. Although we can claim that the expected number of relations included by this method is the same as for the random triple sampling, the structure of the sample differs heavily. For each user that is sampled, we now also include the full set of literals associated to the user, but in expectation there are also more users with no literal at all. This also has an effect on relations between resources, as in many cases, these resources will be linked to nothing else.

Still, such a sampling would enable analyses related to literals and together with its good runtime performance, we could utilise it in the case that low computational power is at hand.

5.3 (*) Random Subgraph Sampling

As mentioned in the previous subsection, choosing random abbreviated blocks as sampling yields a graph with underrepresented interlinkage of resources. To counteract this effect, we propose an extension utilising larger structures than a resource and its neighbours. We choose a set of starting resources at random. For every starting resource, we perform a Breadth First Search (BFS) for incoming and outgoing edges up to a certain depth and include the closure of these edges. When investigating interlinkage, it would be more suitable to attempt to find maximal subgraphs, i.e. not limiting the depth of the BFS. This would capture all maximal paths. However, we cannot estimate a priori how large

such a sample could get and in the worst case – we might obtain almost the full graph, if the complete dataset contains few isolated subgraphs.

The benefit of using this sampling method is that with high probability, we will be able to retrieve results when searching for local patterns, if such a pattern exists in the full dataset. As an example, we could check if two users who commit to the same project tend to also follow each other. With a connected graph of sufficient depth or maximal subgraphs, such relation chains would be present, if there are enough such relations in the full dataset.

To be able to perform a BFS search, we need to query the outgoing and incoming edges of every desired node quickly. This would be trivial, if the whole dataset were loaded into a triplestore. However, at that point in time, we were uncertain if we could achieve this, due to lack of computation power. As triplestores implement multiple features that are not necessary for this task, we developed our own approach, focusing on the basic graph traversal functionality. Our approach is using Apache Cassandra [67], an open-source, wide column store, NoSQL database management system. The design and features of this tool all serve the functionality of fast retrieval of indexed data. For any triple (Subject, Predicate, Object), we would store that Subject has an outgoing edge to Object, and, if the Object has a URI, also that Object has an incoming edge from Subject. The type of relation is irrelevant for BFS and would increase the stored data significantly. Therefore, we end up with a three column table: `URI, refers_to, referred_by`. We integrated this into the converter to run alongside the triple generation process. In order for this to work, the dataset must be free of blank nodes so that we can refer to every resource. For this reason, we added the `-noblank` parameter to our converter, forcing it not to create any blank nodes.

Overall, saving the outgoing edges of resources costs only a little extra time. As the data from GHTorrent is presorted and we are often able to abbreviate many successive triples, this usually allows to add many relations to the outgoing column of a resource in one update step. Saving the incoming edges however proved to be more difficult. As an example, if one project has a large number of commits, it still requires just one query to save all commits as being adjacent to the project. However, we need to insert the project to be adjacent for every commit separately. The Linux GitHub repository has over 800,000 commits as of May 2019 [68], and we even found a repository with a million

commits. [10] Even with the usage of bulk queries, this step caused the run-time of the conversion process to increase so drastically that we were forced to abandon this approach, with an estimated overall run-time of at least several weeks.

Assuming that only the forward direction is available, some sort of BFS would still be possible. A lot of data would be missing though, even if maximal subgraphs are created. While we re-modelled the turtle output to not contain blank nodes, these originally blank nodes still have no incoming edges, but only outgoing. Therefore, they could not be reached, neither any nodes which are "dangling" from these nodes, meaning having only an incoming edge from the originally blank nodes. For example, this is the case for the `programming_language` class in our ontology, which is "dangling" from the `github_project_language` class, that has only outgoing edges. This issue could be addressed by the addition of inverse relations, which we will discuss in Section 10.

5.4 (†) Chronological Sampling

Although Random Subgraph Sampling possesses very favourable properties, the intractable computation time led us to a different approach of creating representative datasets. We realised that the input tables of the converter were ordered after one of the key columns. This means that a newly created user, project, comment or commit would be listed at the end of the input table.

If we assume that all input tables grew with the same ratio until now, we can take a fixed percentage p_{hist} of lines from the beginning of each input table and obtain the RDF containing a complete historic data set up to an unknown point in time. For some input tables containing resources and their connected datatype properties, which are always in one to one correspondence, we know exactly that the historically first $p_{hist}\%$ of data is included. More problematic are tables linking different resources in a one to multiple or multiple to multiple correspondence. As the input tables are sorted according to just one key, the resulting dataset will contain all information about less than $p_{hist}\%$ of resources belonging to first key, but no information for the other keys. Formulated in an example: The sample will contain a link to the newest commit of an old project, because the according input table is sorted by the project IDs. However, no data will be present about this latest commit. Overall, this is just a minor drawback that we need to keep in mind for our analyses, as a sample derived from such a method should still be highly connected.

[10] `https://github.com/cirosantilli/test-many-commits-1m`

If we impose additional but very weak assumptions, we might even derive some tendencies about the grade of connectivity of such a sample. These assumptions are that users, projects, organisations etc. that exist for a longer period tend to have more adjacent relations than new ones. This leads us to the conclusion that a sample from the beginning of the input files is more interconnected and contains less small partitions, than a sample of the same size from the end of the input files. As these two types of samples are the extremes in sense of the distribution of partition sizes, such chronologically sampled `head` and `tail` samples can be used as boundaries for different estimation and evaluations.

Another upside of this sampling method is the ease of implementation. As only the input tables need to be reduced to a certain size or percentage, the converter does not need any additional features. For the selection of a fixed set of rows or percentage of a file, from the beginning, end or middle, multiple shell commands exist in every common operating system. For Linux, we used combinations of the `head`, `tail` and `wc` commands to implement such a sampling.

Like random triple sampling, these new sampling methods also require almost no computational effort in generation, but have some of the desirable properties of Random Subgraph Sampling. In the later sections, we use such samples for the evaluation of loading times and for the design and estimation of query results and performance.

6 (†) Deployment of Server and SPARQL-Endpoint

Content of this section will be the long process of transforming our database from a plain text file to an efficiently queryable endpoint. In Section 3.2, we discussed the approaches of different software providers, which all utilise a superior infrastructure. *Corcoglioniti et al.* already pointed out how difficult the deployment of big data on a single node triple-store is [30]. Nevertheless, we wanted to try a deployment by ourselves.

In Subsection 6.1, we will start with a short discussion about different triplestores. While their capabilities and requirements were already mentioned in Section 3.2, here we will focus on the practical aspects, i.e licenses and comparing benchmarks. Subsection 6.2 will introduce the first attempt of loading the data, its failure and the evaluation methods guiding our following decisions. The remaining parts will chronologically recall our attempted approaches, before we finally obtain a strategy for performing analyses on the dataset that fits our resources for both time and computational power.

6.1 (†) Used Software

Our first research leads to a broad set of software, claimed to have suitable functionality for operating on big data RDF files. Graph databases either contain a native support for RDF or extensions which enable comfortable handling of those datasets. We have set our focus on tools of the first class, dedicated to handling RDF, hoping that such specialised tools might perform more efficiently on large scale datasets. Following this restriction, we elected the following candidates for further research, recommended by the official page of the W3-Consortium: *Oracle Spatial and Graph* [69], *Stardog* [70], *AllegroGraph* [71],*Open-Link Virtuoso* [72], *GraphDB* [73], and *Blazegraph* [74].

Although providing promising results, the first four members of this list only offer a free trial version for a fixed time period, are limited to a fixed number of triples or usage of resources. The licensing of the remaining options of *GraphDB* and *Blazegraph* seemed suitable. While *GraphDB* offers a free version, limiting the number of parallel queries to two, *Blazegraph* is open-source licensed under GPLv2. For the final decision, we considered two major factors: A suitable query performance for the analyses done in this master thesis and the possibilities of maintaining an open-access version of the dataset. For both solutions, there are analyses on which they are fast, while performing slow for others, according to different Benchmarks, comparing one of both tools against other

solutions. [75–77]. A major drawback for this kind of research is the limited literature available comparing both tools in the same setting. While there are some recent contributions by *Georgala et al.* [78] and *Hernández* [79], directly comparing *Blazegraph* and *GraphDB*, the benchmarking goals and used infrastructure differ from ours completely and therefore only have limited significance.

Concerning the opportunity to provide a SPARQL endpoint, both *Blazegraph* and *GraphDB* support the deployment of an easily customisable web interface with options for setting query time limits and a client based hourly query limit. Naturally, *GraphDB*'s free license, limiting the amount of queries that can run in parallel, led us to favour *Blazegraph* in the long-term usage.

Besides the previously discussed software, a new solution appeared during our writing process: *AnzoGraph*. According to the audit log of the W3C website [80], *AnzoGraph* was added to the listing on W3C on the 1st of March 2019, which is long after we started the load in process with *Blazegraph*. Although claiming good results for query and load times, *AnzoGraph* needs four times the storage size of the dataset as a recommended requirement for the memory, which is not maintainable for us at the moment.

6.2 (†) Load-In

For the reasons mentioned in Subsection 6.1, we deployed *Blazegraph* on our server, comprising an instance responsible for the loading of the data and a curl-based client, managing load requests. Our intended workflow was to deploy the raw data and perform our analyses afterwards. We aborted our initial trial to load the whole November 2018 dataset, which is 310GiB in size, into the standard server configuration of *Blazegraph* after 131 hours. Upon reading the size of the journal, the internal storage system of Blazegraph, containing 281GiB stored on the hard-disk, we could estimate that only a subset of 3.3 billion triples, less than 15% of our data, was loaded at this point of time according to *Blazegraph*'s official statistics [81]. Our results from the end of this section even suggest a far lower value. The main motivation behind the abortion was that we couldn't observe any growth of the size of the journal on the hard-drive for more than a day, meaning that the process froze or that the load rate dropped significantly.

This encouraged us to start a series of experiments to determine a suitable server configuration and estimates for the runtime. *Blazegraph* supplies different standard configu-

rations, enabling or disabling various features like reasoning, free-text index or statement identifiers. As our main usage criteria is to analyse the already defined semantic dataset, especially the fastload configuration seemed appropriate, described as [82]:

> *This mode still does inference, but it is database-at-once instead of incremental. It also turns off the recording of justification chains, meaning it is an extremely inefficient mode if you need to retract statements (all inferences would have to be wiped and re-computed). This is a highly specialized mode for highly specialized problem sets.*

6.2.1 (†) Initial Evaluation Plan

Before listing the set of tested configurations, We want to explain the evaluation methodology. As mentioned in Section 5.4, both `head` and `tail` samples seem to be structure preserving and feasible to compute, making them appropriate for the evaluation of load in performance. As we expect that head samples contain more internal linkage than the actual dataset, they provide a better upper bound on the load in times. We derive this from the assumption that a connected graph of a fixed size should not be easier to store for efficient querying than a set of two isolated partitions. An initial test of loading times on the standard server configuration for head samples led to the conclusion that file sizes above 10GiB seemed inappropriate for extensive benchmarking. In fact, we tried to load a sample of 15GiB and aborted the attempt after two days. From our experience, we expected that the load in rate develops highly non-linear and therefore used an exponential scaling, always doubling the size of the previous test sets. These thoughts resulted in a test set of six `head` samples of approximately 320MiB to 10,240MiB for trying to calculate an upper bound on the load times of the whole dataset. An initial test of loading all benchmark sets took 20h:26m in total with 14h:33m for the 10,240MiB sample. Although the loading time seems to be prohibitive for frequently repeated tests, it will become clear in the next parts, that the inclusion of such big datasets as test sets is necessary.

Concerning the test cases, the broad set of options prohibits the evaluation of all combinations of predefined server configurations with its variety of values for sub-options on branching factor, cached triple size and memory. We propose a three staged evaluation.

1. We try the full-feature configuration for a maximal memory usage of 8GB, 16GB, 32GB and 56GB. Although having a capacity of 64GB memory, we had to leave enough memory for the system to continue operating and to allow the operating

system to cache data, reducing the number of read operations required on the disk. We decided against enabling any swap memory for multiple reasons, but mostly to avoid that our HDDs would slow the process down.

2. We alter the best configuration from the first evaluation to disable more and more features, until we get a copy of the fastload configuration. We decided on the exact order in which we disabled the features, according to their relevance for us.
3. We test marginal effects of changing different numerical parameters. As we don't know suitable ranges for all parameters, we alter each parameter slightly, to surveil the impacts on the loading rate.

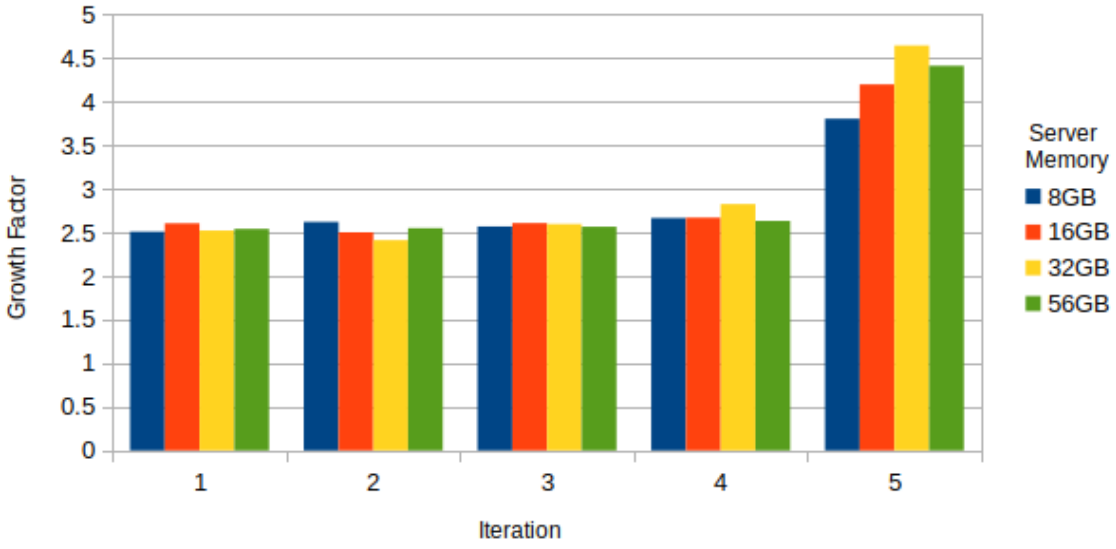
Table 5: Loading times for different sizes of memory

	Memory Size			
Data Size	8GB	16GB	32GB	56GB
320 MiB	310s	305s	308s	304s
640 MiB	777s	794s	776s	772s
1280 MiB	2035s	1985s	1870s	1967s
2560 MiB	5224s	5172s	4852s	5043s
5120 MiB	13919s	13804s	13705s	13262s
10240 MiB	52938s	57920s	63601s	58479s

The table shows the required loading times for 6 data samples of increasing size. These samples were evaluated on 4 different server configurations with different sizes of memory. Memory Size listed in GB, instead of GiB in accordance with common practice for hardware. After each successful loading attempt, each server was reset.

We conducted stage one as intended and obtained the loading times listed in Table 5, providing mixed results supporting our impressions of non-linear growth. By doubling the size of the dataset, the load in time of each server configuration grew with a factor from 2.401 to 4.64. It is remarkable that all server configurations have unexpectedly almost the same growth factor upon doubling the file size up to the point of reaching $5,120$MiB of data with an average multiplication factor of 2.58 for the time and standard deviation of only 0.092, heavily defying our impressions of non-linear growth. The step from $5,120$GiB to $10,240$MiB however shows a growth factor ranging from 3.8 to 4.64 for the different configurations significantly different from the linear growth measured before. The illustration of these circumstances can be found in Figure 7.

To further analyse the growth factor, we decided to move on from the exponentially growing test sets to a finer and linear granularity and evaluate just for a single configuration file, the one with $8GB$ of memory. As this analysis was intended for exploration and not benchmarking against the fastload configuration (data-base at once, for some



The grouped bar plots show the growth factor of loading time for multiple iterations of doubling the sample size. Each server configuration is represented by a single colour and was evaluated for 5 iterations. The data is calculated from the observations in Table 5.

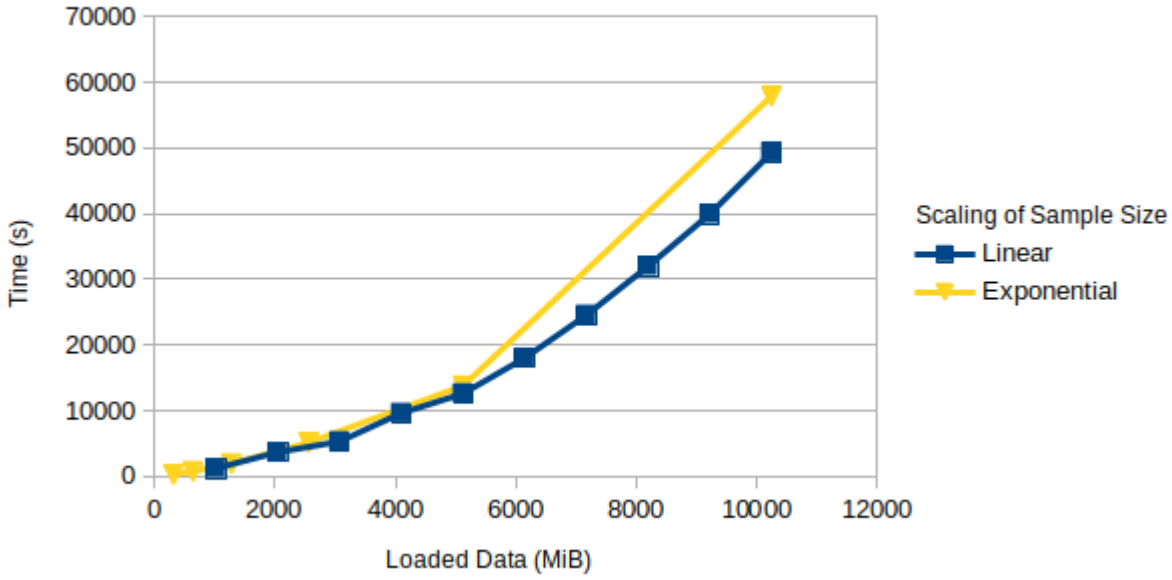
Figure 7: Expansion factor upon doubling the data size

features [83]), we decided to implement an incremental approach, skipping the reset our server and using 10GiB of `head`-sampled data split up to chunks of 1GiB. We hoped that we could save time by avoiding any unnecessary reloading of any data. To ensure that the former 5GiB and 10GiB test files resemble a combination of the first 5 and 10 chunks respectively, we split the input files of the converter rather than the already generated test data sets. The resulting load in times can be found in Figure 8.

Not only do the times depicted show a slower increase for incremental loading, but also the total load time summed up to only 85.12% of the values depicted in Table 5. As such a high difference could not be the result of external effects, we conclude that Blazegraph's loading routine is not suited for bigger files. This insight also led us to a new strategy for loading the data, as the effort of creating a split version of the dataset just requires small modifications to the converter.

6.2.2 (†) Altered Evaluation plan

To evaluate the possibilities of incremental loading of split datasets, we came up with two different approaches. We split our dataset `combined.ttl` into files `part1.ttl`, ..., `partN.ttl` consisting of a (1) fixed size or of a (2) fixed amount of triples. For both variants, a distribution preserving split in the way we implemented it at the end of Sub-



Loading times of different trials. "Exponential" represents the first run with doubling sample sizes per iteration and a server reset after each load. "Linear" describes the second trial with linear increasing sample sizes and an incremental loading.

Figure 8: Loading times with finer samples

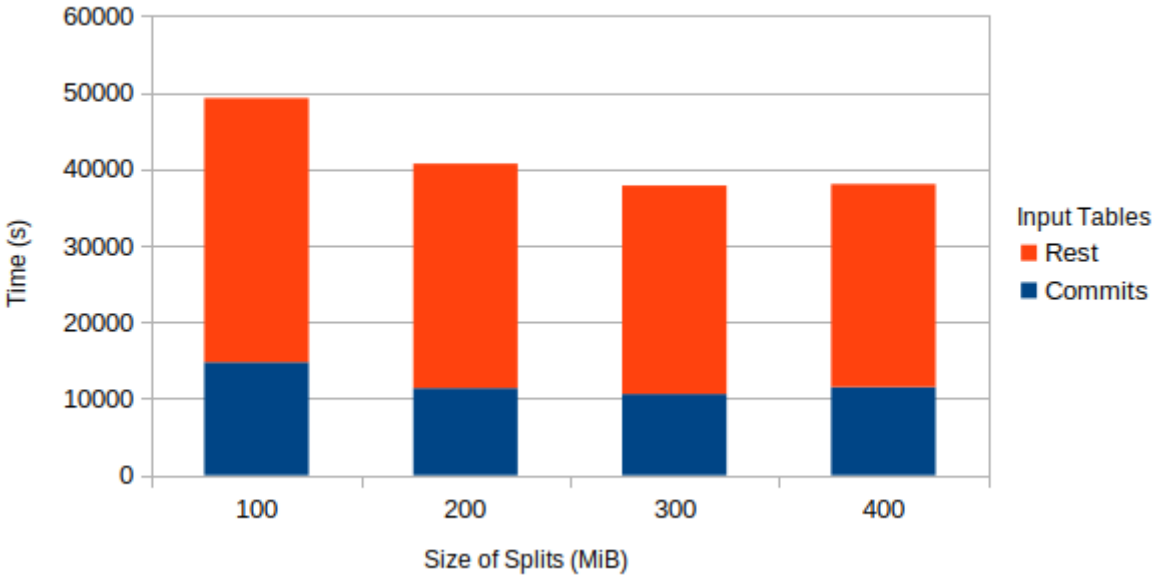
section 6.2.1 seemed too costly. This would require a lot of preprocessing for splitting the raw input files and sorting them according to the distribution of relations. Instead, we modified the converter to generate upon receiving an input table like "parents" a distinct set of files `parents_part1.ttl, parents_part2.ttl, ..., parent_partN.ttl`, which then could be combined to form a test sample.

Out of the two variants, we chose to implement the first. The second variant of a fixed amount of triples had major drawbacks. As long as we link different entities or literals with a fixed type, like `xsd:dateTime`, the required storage of a triple remains almost fixed, because it contains only prefixes of up to two characters, IDs and literal values of similar length. Considering strings for representing comments, this statement is unfortunately not valid anymore, as a comment's length is not predictable and can range up to several ten thousands of characters, which could skew the distribution of information content for each sample. The only design question that remained to cope with is the parameter for the file size. To determine a suitable one, we altered the evaluation plan of step 2:

2. (a) Split the 10GiB sample set into files of fixed size to determine the best parameter for the file size
   (b) Evaluate the full feature configuration for split files against the fastload configuration for non-split.

For the first part, we choose file sizes starting at 100MiB in steps of 100MiB, until no improvements could be observed. This was the case for the step from 300MiB to 400MiB. A graphical representation of the results is provided in Figure 9. The bars show the overall loading times, and also the loading times of the commit data, the most common data type in our dataset. The first thing to point out is that the load in times for 100MiB splits, is slightly faster than loading everything at once. The following steps to bigger split sizes show improvements from 100MiB until 300MiB by a reduction of 23.21% of the loading time, totalling to 28.51% less time consumption compared to the initial "everything at once" loading approach. Going forward, a slight increase for 400MiB is observable which fits to the previously observed improvement of 14.88% for splits of 1GiB in size. Seemingly we have a trade off, that smaller file sizes lead to a faster loading, but also cause computational overhead when getting too small.

We choose for all future load-ins sizes of $300MiB$, as they provided, at least for the 10GiB sample, the best trade-off and therefore hopefully also for the whole data set.

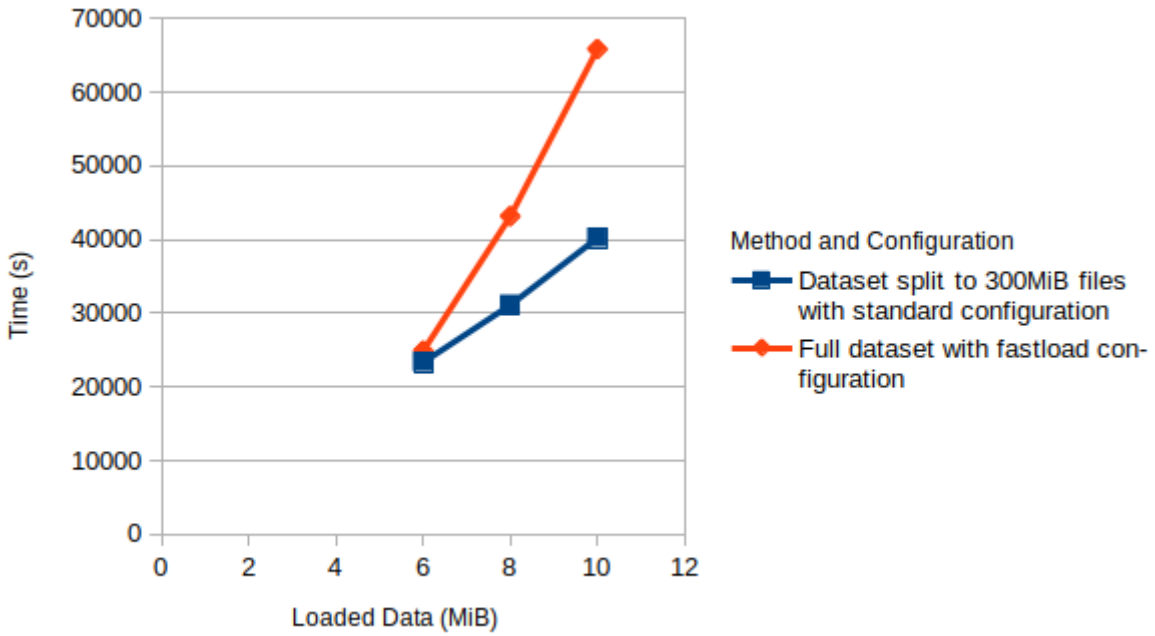


Loading times for a dataset of 10GiB, split up into several files. The experiment was conducted for split sizes from 100MiB to 400MiB. The loading times for data originating from the commits table is separately coloured in blue, as they form the largest share of the data.

Figure 9: Loading times for different split sizes

In part (b) of step two, we finally compare against the fastload configuration, proposed by the Blazegraph documentation. From the previous results, we know that significant differences are observable after 5GiB of data size. For this evaluation, we again choose a

linear scaling, to obtain a better intuition of the growth behaviour beyond 10GiB. The chosen test sizes were 6GiB, 8GiB and 10GiB and we evaluated two different routines. The fist one is the previously discussed loading with split data combined with the standard server configuration. The second option is Blazegraph's fastload configuration, which disables features like automatic reasoning and inference. As the fastload configuration implements a "database at once" approach, the full dataset was used for this method. Surprisingly, the standard configuration with split data beat the fastload configuration in every case, being up to 39% faster. Seemingly the fastload configuration would outperform all other methods for small data sizes, but also shows a steeper growth when using bigger datasets. These final results can be found in Figure 10.



The loading times for the two resulting approaches. Sample sizes of 6GiB, 8GiB and 10GiB. Axes are not truncated to visualise the magnitude.

Figure 10: Loading times Comparison

This was the final evaluation step, as we abandoned stage 3, the finetuning of parameters in Blazegraph. A single test of altering the parameters as described by the Blazegraph team [84], led to longer loading times of 7% for the 10GiB sample. As we had no better starting point for finding a good parameter configuration, we decided that successive tests to find the optimal combination of these 8 values are too costly.

6.3 (†) Our Resulting Approach

It is not determinable from the previous evaluation, which configuration is the best one for our case. As already mentioned, the fastload configuration is faster but has a steeper growth. Fortunately, we realised that the improvements that we achieved with the loading of split data is sufficient enough for deploying a new strategy. We realised that even with all our optimisation efforts, the loading of the whole dataset is unrealistic, but also not necessary. For most of our planned analysis, it should be enough to focus on certain subsets of the dataset.

8GiB of memory should be sufficient for any loading process, according to our own evaluation. As a result we deployed a Blazegraph instance with 8GiB of memory alongside an instance for querying with 48GiB of memory on the same server. Each instance would operate on a separate hard-disk of 4TB. The first server would be responsible for loading the data incrementally. Whenever sufficient data for performing desired queries is loaded by the first instance, we copy the whole journal into the second instances file path and run the analysis in parallel to the data loading process. With a measured size expansion factor of around 10, both instances could theoretically contain the whole dataset as a Blazegraph journal. We ordered all analyses we would like to perform by the effort required to load the needed data and obtained an order for loading different subsets of the RDF.

1. Users, Followers, Organisation Members
2. Project Members, Project Join Events, Programming Languages, Watchers
3. Projects, Pull Requests

...

We still consider the effort of loading `commits`, `commit_parents` and `project_commits` as too high. The three tables need 143GiB, 21GiB and 44GiB of storage respectively, summing up to 67% of data that we have. As of the start of June 2019, where we aborted any further loading, the final Blazegraph journal contained [11]:

Triples: 2,717,664,071
Entities: 245,069,646
Properties: 42

Considering that the journal size of this dataset already reached 456GiB, we can correct the estimates from our first loading approach down to only 1.6 billion triples within a

[11] The description file was uploaded to: `www.semangit.de/summary.rdf`

week. With the fact in mind, that the growth rate of loading time is highly non-linear, we are really proud that we manage to store as many relations, especially with specifications significantly lower than the suggestions [81]. Moreover, our dataset is actually queryable in feasible time and the analyses conducted in Section 7 are not based on samples, but on real data.

We can summarise that we performed better than expected. While we now know how to fasten the load in process, we still need a better and more efficient architecture if we want to deploy the full dataset or perform analyses that require larger data, such as commits.

7 (†) First Analyses on top of the Graph Database

We described our third goal as the task of finding use cases and evaluating the query capabilities. In this Section we will discuss three example use cases in detail, provide SPARQL queries for them and show first results. It is important to note that these analyses do not operate on samples. As described in Section 6, we were not able to deploy the full dataset, but at least the complete information about 42 properties and the associated classes. This dataset, containing 2.7 billion triples, was used for the following results. We already listed different ideas and analyses that were implemented on data sources similar to ours in Section 3. In this Section, we will conduct our own analyses, mainly focusing on tasks utilising more complex graph traversals.

In Subsection 7.1, we show how users cooperate on an international basis. We investigate the number of nationalities that one user of a distinct country collaborates with and also show how our data can be utilised to detect a good collaboration between two nations. Thereafter, in Subsection 7.2, we will extract the number of followers within organisations and measure the success of their projects by counting their watchers. With this information, we investigate if more successful organisations share a more interlinked structure within their teams. In Subsection 7.3, we revisit our language analysis from the motivational section from a different point of view, with a graph database at hand. Further analyses that require more computational power or sampling will be discussed in Section 9.

7.1 (†) Global Cooperation within Repositories

Many users on GitHub state the country they live in or where they originate from. With this data, we can derive interesting analyses for countries on a global scale, enabling comparison on regional differences for programming languages, coding style, social media behaviour or even business policies. We already mentioned publications analysing such geographic information, for example *Rastogi et al.* that analysed pull request acceptance behaviours for different nationalities [54]. Besides comparing differences regarding those aspects, one can also analyse how well countries cooperate. Especially such an analysis could be interesting for governments, trying to support their IT sector in the internationalisation process.

A repository can have multiple collaborators. The SemanGit dataset represents this information through the class `project_join_event`, linked to a user and a project. An intuitive approach to obtain data about international cooperation is to query each project that contains at least one user from a certain country, and count the nationalities of the cooperators:

Using the server described in Section 6 and the most recently loaded version of the

```
-- Count the set of users from countryA, that work
-- in at least one project with a user of countryB.
PREFIX sgo: <http://semangit.de/ontology/>
SELECT ?countryA ?countryB (COUNT(distinct ?userB) AS ?relations) {
 ?userA sgo:github_user_country_code\/ ?countryA .
 ?projectJoinEvent1 sgo:github_project_joining_user\/ ?userA ;
                    sgo:github_project_joined\/ ?project .
 ?projectJoinEvent2 sgo:github_project_joined\/ ?project ;
                    sgo:github_project_joining_user\/ ?userB .
 ?userB sgo:github_user_country_code\/ ?countryB .
 FILTER ( ?countryA != ?countryB )
} GROUP BY ?countryA ?countryB
```

Query 1: Querying the number of users of a country, that work with at least one user of the other country on the same project

dataset, which is November 2018 at the point of writing, this query took 31h37m to compute and resulted in a list of 11,442 combinations of countries and their respective count statistics, which was agglomerated of 30,101,550 observations of `(countryA, countryB, userB)`. To improve readability, we use alternative names for the representation of these variables. Let c_A, c_B denote `countryA`, `countryB` respectively, and $r(c_A, c_B)$ the number of cooperations between these two countries as counted in the query.

In Table 6, the top 10 entries are listed, sorted by $r(c_A, c_B)$. Observable is a strong bias for a good international cooperation of users from the United States of America (USA) to other nations. To obtain more meaningful results, we decided to normalise $r(c_A, c_B)$ by the number of GitHub users $p(c_B)$ from c_B. The query responsible for counting the number of users for each nation, took 23m to compute. We combined both datasets in the post-processing and obtained the results described in Table 7. The results are heavily skewed, which is due to a low number of users from certain countries. There is only one user from Norfolk Island and only five users from the Democratic Republic of Congo. If just one user of these countries has worked on a project as big as the Linux Kernel, it is sufficient to move that country into the top 10. We therefore censored our data by delet-

ing all countries with less than 1,000 reported developers and recalculated our statistics, shown in Table 8.

Table 6: Absolute number of collaborations between nationalities

c_A	c_B	$r(c_A, c_B)$
gb	us	18,760
ca	us	17,346
de	us	13,881
au	us	10,395
in	us	9,320
nl	us	8,635
fr	us	8,180
us	gb	7,246
jp	us	6,246
es	us	5,980

The absolute number $r(c_A, c_B)$ of users from country c_B that work with at least one user from c_A in a shared project. The top 10 observations are listed.

Table 7: Average number of collaborations between nationalities

c_A	c_B	$r(c_A, c_B)$	$p(c_B)$	$avg(r(c_A, c_B))$
au	nor	1	1	1.00
cd	nor	1	1	1.00
gb	nor	1	1	1.00
gb	cd	2	5	0.40
au	cd	1	5	0.20
nor	cd	1	5	0.20
dk	cd	1	5	0.20
us	cd	1	5	0.20
mg	cd	1	5	0.20
za	cd	1	5	0.20

The average number $avg(r(c_A, c_B))$ of users from country c_B that work with at least one user from c_A in a shared project. The top 10 observations are listed.

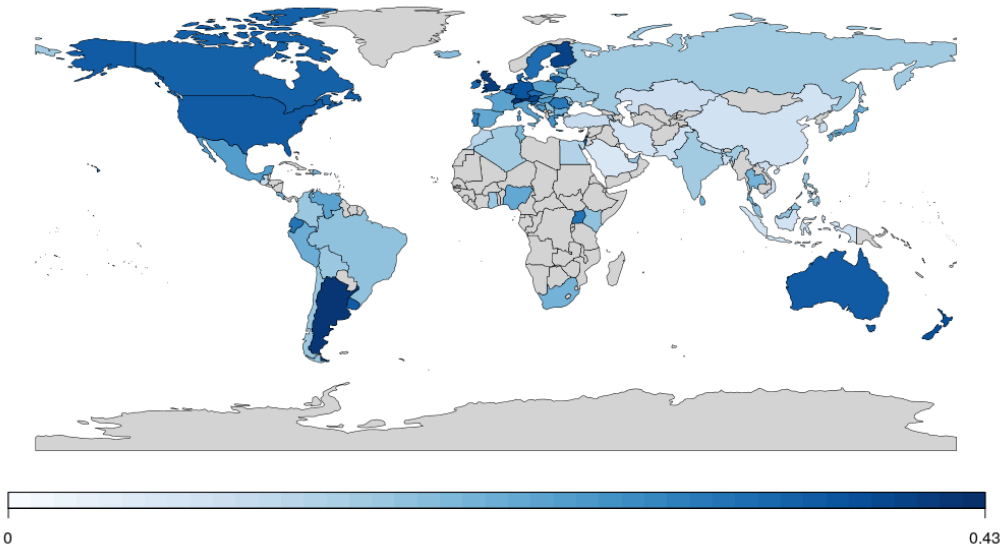
Table 8: Absolute and average number of collaborations between nationalities

c_A	c_B	$r(c_A, c_B)$	$p(c_B)$	$avg(r(c_A, c_B))$
us	ca	5,766	78,273	0.074
us	il	511	7,908	0.065
us	gb	7,246	122,341	0.059
us	uy	130	2,252	0.058
us	no	655	12,884	0.051
us	lk	168	3,311	0.051
us	ch	932	18,473	0.050
fr	lu	55	1,102	0.050
us	au	2,305	46,635	0.049
us	ar	727	15,030	0.048

The average number $avg(r(c_A, c_B))$ of users from country c_B that work with at least one user from c_A in a shared project. The top 10 observations are listed, after censoring nations with less than 1,000 developers.

By this measure, we find that many nations collaborate with a huge amount of USA citizens. The only nation that collaborates more with another nation than the USA is Luxembourg, sharing a large cooperation with France. As the citizens of the USA form the largest group within the GitHub users with a share of 29.38%, it is more likely that users work on the same project with a USA citizen, than with any other country. As

this type of analysis seems to be very one-sided, we will move to a different approach. Namely, we will sum up $avg(r(c_A, c_B))$ over the variable c_A to obtain the average number of nations a user of c_B is working with. These results are depicted in Figure 11.



The shades of blue encode how many other nationalities an average user from the coloured country is working with. Nations with less than 1,000 users are excluded and coloured in grey.

Figure 11: Average number of nationalities a user is working with

In the top 5 of most internationally working countries are Luxembourg (0.435), Norway (0.434), Argentina (0.419), Switzerland (0.418), and Great Britain(0.400). On the other side of the scale, we find countries like Saudi Arabia (0.064), Indonesia (0,081), China (0.084), Pakistan (0,085) and Turkmenistan (0.086).

Although a huge proportion of users from many countries works mostly with developers from the USA, the USA themselves do not have the highest average number of nations they are working with. We therefore ask ourselves if there are pairs of nations, that share both a high value of $avg(r(c_A, c_B))$ and $avg(r(c_B, c_A))$. Unfortunately, the construction of a scoring function that assigns higher values to observations where both values are high, but penalises huge differences, is a trade off problem. We will use a simple scoring function $p(c_A, c_B)$ and the index $h_{A,B}$ which is the normalisation of that scoring function.

$$p(c_A, c_B) = \frac{avg(r(c_A, c_B)) + avg(r(c_B, c_A))}{2} - |avg(r(c_A, c_B)) - avg(r(c_B, c_A))|$$

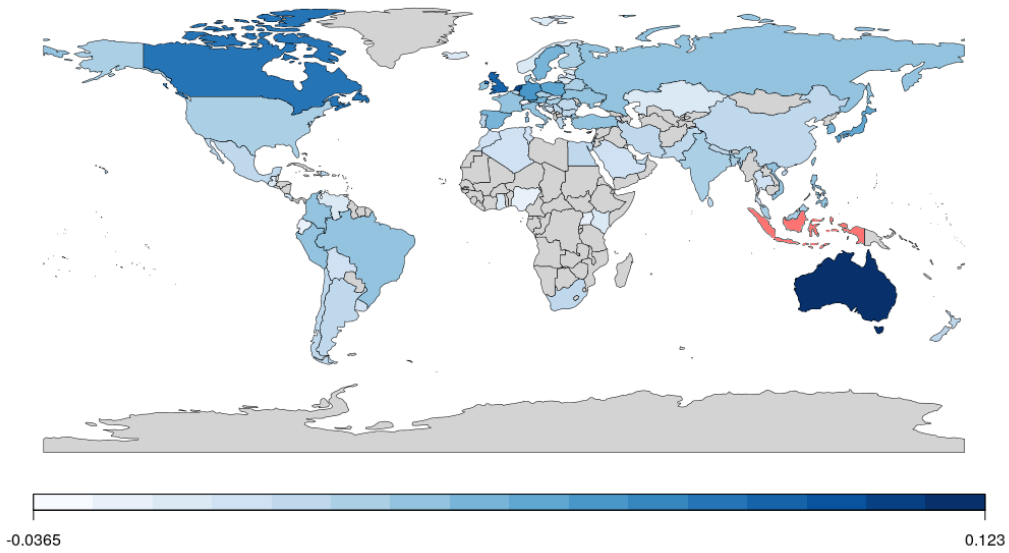
$$h_{A,B} = \frac{p(c_A, c_B)}{\max_{A,B}(p(c_A, c_B))}$$

We compute this index for all countries that we have enough observations about, i.e. at least 1,000 GitHub users who state it as their location. The results for the top 5 index values are depicted in Table 9. We can now restrict our data to a certain country c_A and obtain information about which other countries have good symmetric collaborations. An example of such an analysis is provided in Figure 12 for the case of Indonesia, showing good connections to Australia, Canada and Great Britain, but also to the Netherlands, which used to have a colony in Indonesia. We conclude that, although this methodology still requires further development, the current combination of query and statistics provides a possible starting point for further investigation.

Table 9: Index $h_{A,B}$ for international cooperation

c_A	c_B	$avg(r(c_A, c_B))$	$avg(r(c_B, c_A))$	$h_{A,B}$
de	gb	0.0238	0.0241	1.0000
ca	gb	0.0212	0.0178	0.6800
ca	fr	0.0151	0.0143	0.5904
au	gb	0.0247	0.0171	0.566
de	fr	0.0142	0.0159	0.5656

Top 5 observations for the index $h_{A,B}$, measuring a bi-directional high cooperation of c_A and c_B.



The shades of blue encode which nation has a high index of cooperation $h_{Indonesia,B}$. Nations with less than 1000 users are omitted in grey.

Figure 12: Index values for Indonesia

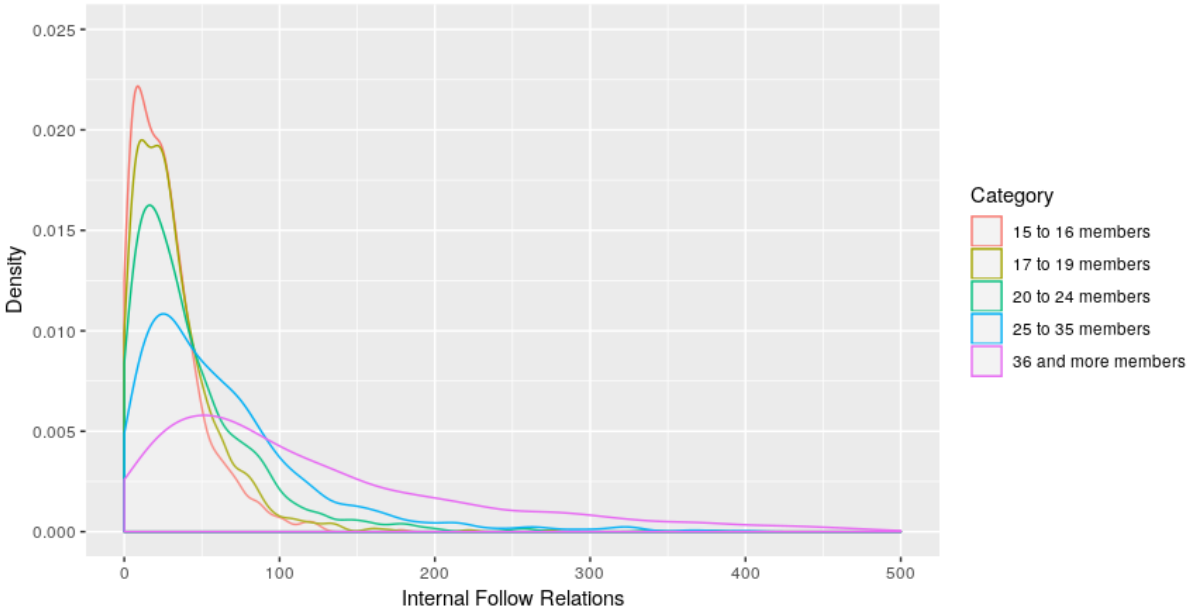
7.2 (†) Social Relations in Organisations

We already found multiple publications, regarding how social mechanisms can help to promote a project and ensure its long term sustainability and survival [44, 48, 49]. We want to follow a similar approach and try to investigate if successful projects have a different internal social structure. Namely, we want to analyse the follow relationships within organisations. To get an initial idea about the scale, we queried all organisations on GitHub and extracted the number of members together with the number of follows within them.

```
--Querying organisations and count their users
PREFIX sgo: <http://semangit.de/ontology/>
SELECT ?organization (COUNT(DISTINCT ?user1) AS ?users)
       (COUNT(distinct ?follow_event) AS ?follows)
{
 ?organization sgo:github_user_is_org true .
 ?join_event_1 sgo:github_organization_is_joined\/ ?organization ;
               sgo:github_organization_joined_by\/ ?user1 .
 - Collect the internal follow relations
OPTIONAL {
    ?join_event_2 sgo:github_organization_is_joined\/ ?organization ;
                  sgo:github_organization_joined_by\/ ?user2 .
    ?follow_event sgo::github_follows\/ ?user1 ;
                  sgo::github_follower\/ ?user2 .
    }
} GROUP BY ?organization
```

Query 2: Counting users and follow events for organisations

To get an impression of the relation between the number of users and the organisation's internal follow relationships, we decided to divide the results into subgroups. We excluded all organisations with less than 15 members, leaving only 4,736 out of 245,123 observations in the dataset. Thereupon, we grouped the remaining observations into 5 groups of nearly equal size, ranging from 931 to 982 observations. The distribution of follow relationships per group is shown in Figure 13, together with the resulting groupings. This visualisation contains some interesting insights. Seemingly, all groups have the same class of distribution, heaving a lot of projects with small internal follow relations and increasingly less density. In addition, all distributions seem to be bi-modal, whereby the second bulge seems to move to higher values on the x-axis. Of course these statements need a proper statistical analysis to be verified. Nevertheless, we are interested in the difference between successful and unsuccessful organisations, where a bi-modal distribution



The figure shows the density plots for 5 categories of organisations, measuring the number of follow events within the organisation. The dataset is censored to contain 500 follow relationships at maximum to remove outliers

Figure 13: Density plot for organisation internal relationships

might be a good indicator.

For obtaining the classification between good and bad, we constructed a second query for our dataset. For each organisation, we query the number of watchers for their projects and sum these values. This will lead to a naive measure of a project's popularity. The resulting data can now be matched. For each of the above mentioned groups, we construct two subgroups *less successfull* and *more successfull* by splitting the projects above and below the median. As both subgroups are equal in size size, we can again use a density plot for comparison.

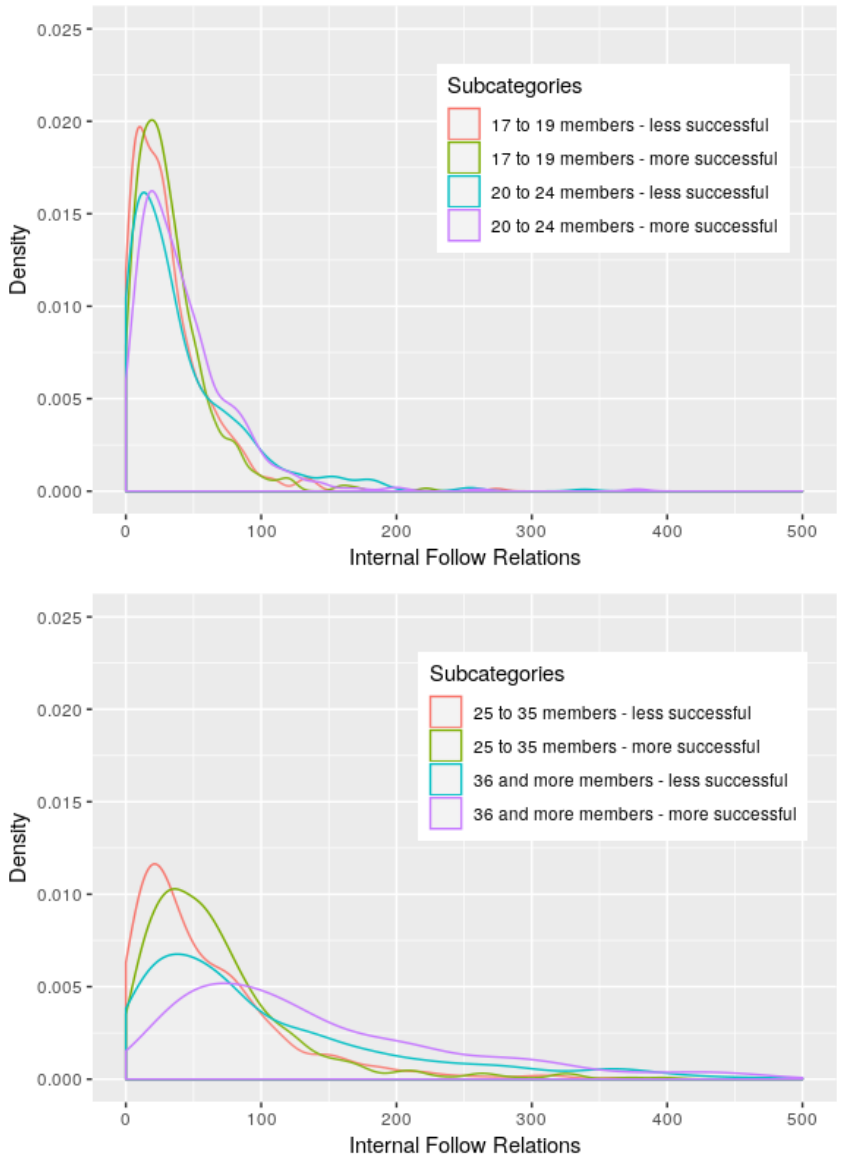
We illustrate the results in Figure 14 and find, that for small organisation sizes, the

```
PREFIX sgo: <http://semangit.de/ontology/>
SELECT ?org (COUNT(?watchEvent) AS ?watchers){
 ?org a sgo:github_user\/ ;
      sgo:github_user_is_org\/ true .
 ?project sgo:github_has_owner\/ ?org ;
          a sgo:github_project\/ .
 ?watchEvent sgo:github_follows\/ ?project .
} GROUP BY ?org
```

Query 3: Counting the number of watchers per organisation

distribution of internal follow relationships and success seem to be uncorrelated. This effect changes if we analyse organisations having more members. Not only that the density

is lower at the peak, but the peak is also located at a higher x-value and we observe a higher number of follow relations. One reason for this effect might be, that our categorisation includes a higher range of users per organisation for the Categories *25 to 35 members* and *36 and more* members. If the bigger organisations tend to be more successful in general, we could observe the same effect of more internal following. This is not the case for our data, where the averages are close to each other. In fact, for the Category of *25 to 35 members*, we found an average difference of 0.27 users between more and less successful organisations. This leads us to the conclusion, that success is correlated with the amount of organisation members, that follow each other, at least for larger organisation. One hypothesis to explain this is, that in such big organisations the ability to follow other developers might implement an additional information exchange between departments that are normally not working together.



The figure shows the density plots for 4 categories of organisations, measuring the number of internal follow events within the organisations. The 4 categories are divided in the subcategories depending of their success, which is measured by the number of watchers for their projects. The dataset is censored to contain 500 follow relationships at maximum to remove outliers

Figure 14: Density plot for organisation internal relationships

7.3 (*) Revisiting Evolution of Programming Languages

Having already analysed the evolution of programming languages, we will now make use of the graph database structure to show further analyses that would have been less efficient to compute with the relational model. As described in Section 2, a language timeline is not a suitable type of analysis for the data we have, as the update quality is insufficient. We therefore propose two different analyses, focusing on locations and popularity instead of time. We start with the presentation of another analysis on the usage of programming languages, this time comparing how frequently the languages are used in various countries. Afterwards, we will look at the impact of projects using certain languages, which is measured by the number of users interacting with them.

7.3.1 (*) Regional Differences in Language Usage

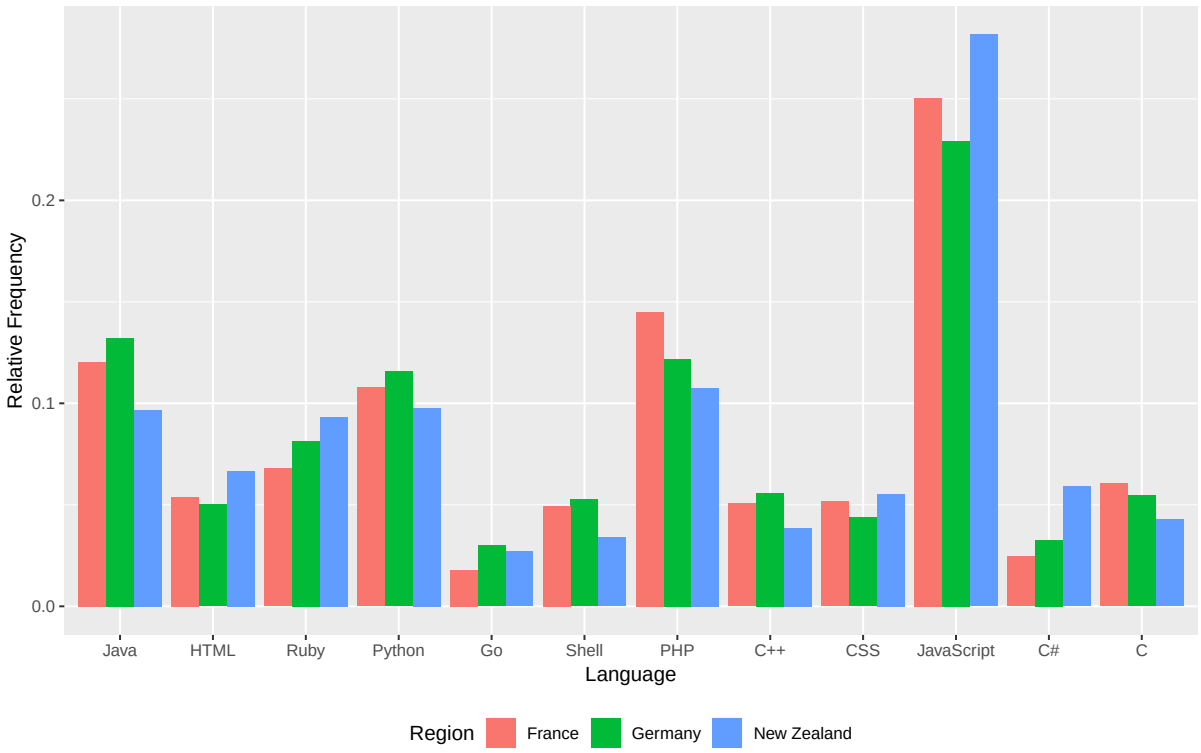
While we previously investigated global trends in the usage of languages over time, we will now analyse regional differences in programming language utilisation. As before, we determine the primary language of a project as the language that makes up the largest portion of the source code. If we assume that it is unlikely for a project to change its primary language, the poor update quality only has a minor impact on our results. Figure 1 showed that the number of projects changing the primary language in years 2015 to 2016, where we observed the highest number of changes in our dataset, is below 100,000, while there were between 20,000,000 to 30,000,000 projects on GitHub at that time [85].

In Figure 15, we compare the primary languages of GitHub repositories from France, Germany and New Zealand. We assign a repository to a certain nation, if the owner of the repository has entered a location within that country in his profile. Note that GHTorrent geocoded the *Location*-field of user profiles, see Section 8.2. As the query of this evaluation is similar to Query 4, we omit its listing here. To be able to compare different countries, we selected the top 12 programming languages of each country, counted the occurrences of those languages as primary languages on repositories of that country, and normalised the resulting values to add up to 1 per country . We refer to this as the relative frequency vector $rel_{country}$. Let $||(x_n)_n||_p = (\sum_{i=1}^{n} |x_i|^p)^{\frac{1}{p}}$ denote the l^p norm. Then we get:

$$||rel_{France} - rel_{Germany}||_1 = 0.124$$

$$||rel_{France} - rel_{New\ Zealand}||_1 = 0.234$$

$$||rel_{Germany} - rel_{New\ Zealand}||_1 = 0.237$$



A comparison of programming languages for France, Germany and New Zealand, measured by the relative frequency of usage per country. For the 12 most used programming languages, each occurence as primary programming language in a repository was counted.

Figure 15: Programming languages for France, Germany and New Zealand

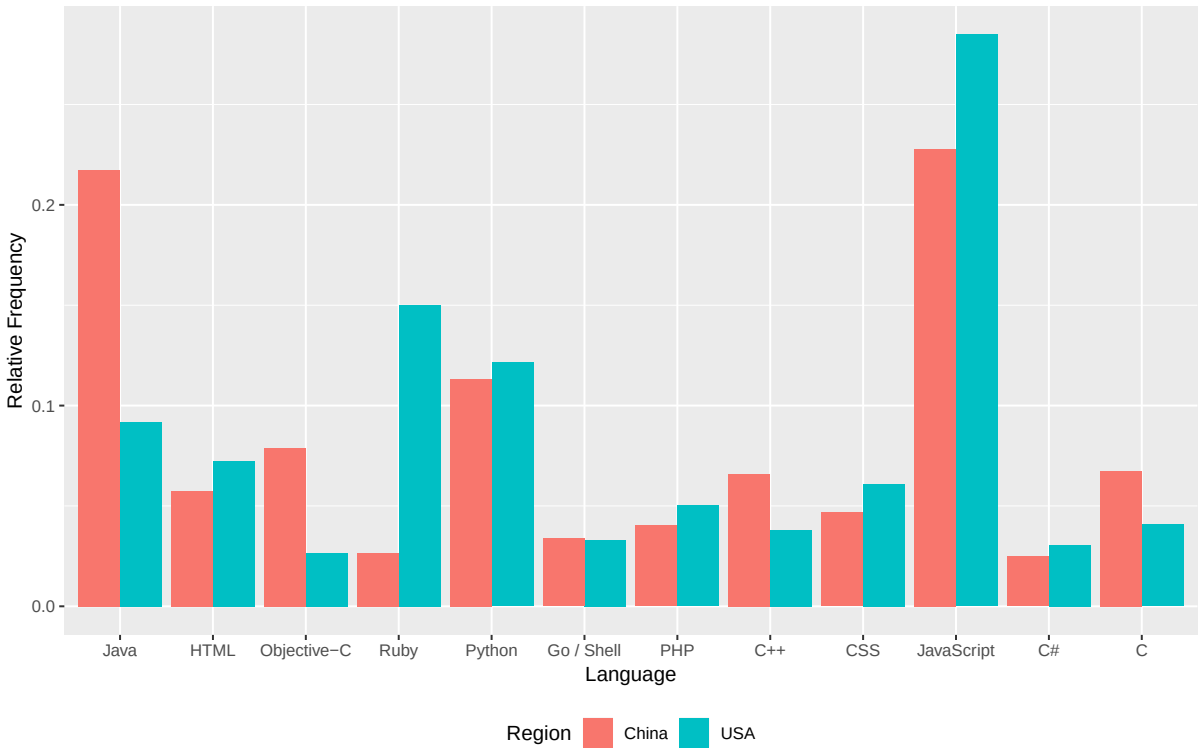
For $p = 2$, we get:

$$||rel_{France} - rel_{Germany}||_2 = 0.001739$$

$$||rel_{France} - rel_{New\ Zealand}||_2 = 0.005845$$

$$||rel_{Germany} - rel_{New\ Zealand}||_2 = 0.002741$$

We used the l^1 and l^2 norms to generate statistics about the differences in distributions. While Germany and France seem to have a rather similar distribution, with a sum of absolute difference of 0.124, the distribution of New Zealand is relatively different to both of these countries. Most notably, New Zealand appears to have a stronger focus on web development and comes first on JavaScript, CSS and HTML, but last on Java, C++ and C. Nonetheless, New Zealand has a higher percentage on C# than the two European nations.



A comparison of programming languages for China and the United States of America, measured by the relative frequency of usage per country. For the 12 most used programming languages of each country, each occurrence as primary programming language in a repository was counted. As Go and Shell both only appear in the top 12 of China and USA respectively, they are directly compared against each other.

Figure 16: Programming languages for China and the USA

To showcase an even higher value of inequality, we have created a second analysis of the same kind. A comparison of China and the United States of America is shown in Figure 16. As before, we compute:

$$||rel_{China} - rel_{USA}||_1 = 0.467$$

$$||rel_{China} - rel_{USA}||_2 = 0.039$$

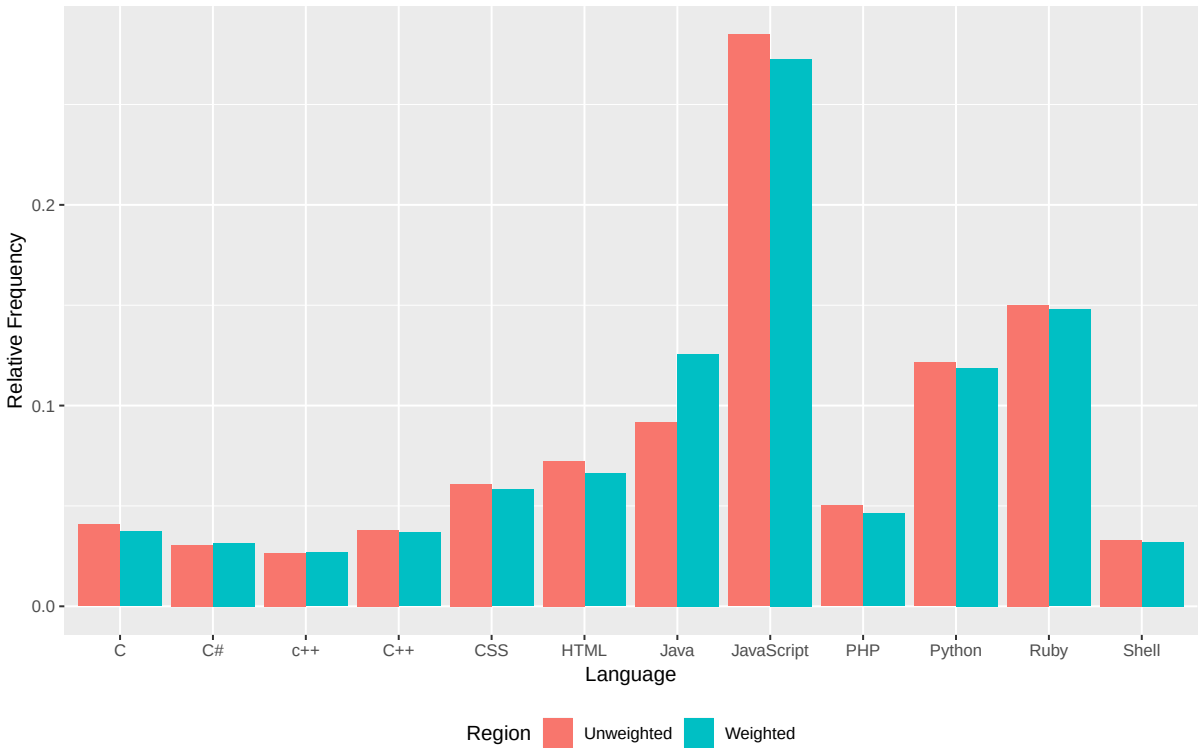
Not only do their most frequent 12 languages differ, some languages even have a difference in relative frequency of a factor of 2 or more. Note that we compared the languages Go and Shell against each other, as Go is not in the USA's 12 most frequent languages, whereas Shell is not in China's. This slightly lowers the distance in frequency vectors computed above. The origin of these popular programming languages seems to have little effect on where it is used. For example, Ruby was invented by *Yukihiro Matsumoto* in Japan [86] and is much more popular in the USA compared to China. Java on the other hand was designed by *James Gosling* from Canada and is developed by *Oracle Corporation* from the United States [87], but it is more popular in China.

Querying the number of occurrences of languages as primary language in a country from our Blazegraph server yielded some surprises with regards to query evaluation time. At the beginning, the time taken for querying about a country like Germany took about five hours. However, querying about China after querying about the USA took less than 30 minutes, even though the result set is much larger. This could potentially be due to Blazegraph storing intermediate results for optimisation.

```
--Querying projects from the United States,
--their primary language and their collaborators
PREFIX sgo: <http://semangit.de/ontology/>
SELECT ?project ?name (COUNT(?joinEvent) AS ?collaborators) {
{
  SELECT ?project (MAX(?langSize) AS ?maxSize)
  {
    --This is evaluated first, as SPARQL works bottom-up
    ?owner a sgo:github_user\/ ;
           sgo:github_user_country_code\/ "us" .
    ?project sgo:github_has_owner\/ ?owner .
    ?language sgo:github_project_language_repo\/ ?project ;
              sgo:github_project_language_bytes\/ ?langSize .
  } GROUP BY ?project
}
--project ?project is already bound from inner query
?currentLanguage sgo:github_project_language_repo\/ ?project ;
                 sgo:github_project_language_bytes\/ ?curSize .
--Accept only primary language
FILTER(?curSize = ?maxSize) .
--Grab reference to "language node", containing the name of language
?currentLanguage sgo:github_project_language_is\/ ?lang .
?lang sgo:programming_language_name\/ ?name .
--Grab collaborators. Note that the owner never "joins" the project
OPTIONAL {
  ?joinEvent sgo:github_project_joined\/ ?project .

} GROUP BY ?project ?name
```

Query 4: SPARQL query to obtain projects from USA, including primary language and weights

In Section 2, we were unable to use the same weight function as GitHub does in their analysis [16]: In their analysis, the weight of a project is the number of unique collaborators. With the graphical model, we are able to compute this weight with little additional effort, as shown in the SPARQL query in Query 4.



Comparison of primary project languages for USA with and without weighting. For each of the 12 listed programming languages, the relative frequency was computed. The number of unique collaborators is used as weighting function.

Figure 17: Weighted and unweighted primary project languages

Figure 17 shows the relative frequency of programming languages occurring as primary language for projects in the USA, both with and without this weighting . For most languages, we only found marginal effects. The highest difference is for Java, which is increased by a large margin by the weight function, suggesting there are many projects with larger numbers of collaborators using Java as programming language in the USA. The strong link between the web languages JavaScript, PHP and HTML is visible in Figure 17 as well, as all of them decrease by a similar factor.

7.3.2 (*) Reach of Programming Languages

We will now drop our geographical focus, where we compared usage of programming languages based on differences in location, and move on to a different type of analysis. More precisely, we are interested in the number of people "reached" by a project featuring a certain language. For example, one could investigate the number of users watching a project, how often a project is forked, starred or contributed to with external pull requests. The canonical measurement of comparing download numbers is not possible for us, as it is not contained in the data from GHTorrent. We chose to use "number of watchers" as popularity indicator, as the data we need to load into our triplestore for this analysis closely

matches the data required for our social analyses, avoiding expensive additional load-ins. The other methods can be performed analogously with only minor modifications to the SPARQL query.

By adding weights to projects based on their reach instead of their size, we strive to analyse a different aspect of popularity. Using size-based weights, such as the number of collaborators of a project, yields an indicator of how popular programming languages are for developers to make software. When using reach-based weights, we gain an insight on how popular the developed products are when made in a certain language.

The query to obtain the watchers for every project, alongside the name of the primary language, is shown in Query 5. As we are interested in the reach of languages, we discard projects with less than five watchers, also to reduce the run-time of the query. Executing this query took 76 hours, which is multiple times more than other queries featured in this section. That is mostly due to the fact that we consider a larger pool of projects. However, this execution time is to be treated with caution for several reasons. Firstly, execution times can be affected by previous queries, as Blazegraph stores intermediate results. Seeing that the queries from Subsection 7.3.1 were executed prior to this query, the run-time is most probably even longer on a fresh setup. Secondly, to evaluate this query, the triplestore required more than the 64GB of RAM that our machine possesses, forcing the usage of Swap space, where hard drive space acts as virtual RAM. This slows down the process, as read- and write operations on the hard drive require more time.

Query 5 does not return a fully aggregated result. Instead, it yields a result in the form of $(primary\_language, \#watchers)$ for every project with at least five watchers. This enables us to dynamically increase the minimum number of watchers required, comparing trends in language of projects with different popularity. Figure 18 illustrates a comparison of primary languages of projects, weighted by the number of watchers. This is done for four sets of projects: those with at least 5, 50, 500 and 5,000 watchers. For the sake of legibility, only the twelve most popular languages (with respect to number of watchers) are considered. As before, JavaScript is undoubtedly the most popular language, with a share of 30.6% for the set of projects with at least five watchers. The second most popular language for the same set is Java, with a share of 12.3%. However, if we only consider very popular projects with at least 5,000 watchers, this result is even more pronounced: JavaScript reaches 39.4%, Java remains second place but drops down to 11.2%. Statistics

```
PREFIX sgo: <http://semangit.de/ontology/>
--Get language of project, including number of watchers
SELECT ?langName ?watchers {

    SELECT ?project ?watchers (MAX(?langSize) AS ?maxSize)
    {
      {
        --Get all projects with at least five watchers
        SELECT ?project (COUNT(?watchEvent) AS ?watchers) {
          ?project a sgo:github_project\/ .
          ?watchEvent sgo:github_follows ?project .
        } GROUP BY ?project HAVING(COUNT(?watchEvent) > 4)
      }
      ?language sgo:github_project_language_repo\/ ?project ;
                sgo:github_project_language_bytes\/ ?langSize .
    } GROUP BY ?project ?watchers

  ?currentLang sgo:github_project_language_repo\/ ?project ;
               sgo:github_project_language_bytes\/ ?myLngSize .
  --Only accept primary language of project
  FILTER(?myLngSize = ?maxSize) .
  --Grab object containing information about the language
  ?currentLang sgo:github_project_language_is\/ ?myLngObject .
  ?myLngObject sgo:programming_language_name\/ ?langName .
}
```

Query 5: SPARQL query to obtain number of watchers for every programming language

on the four sets are shown in Table 10.

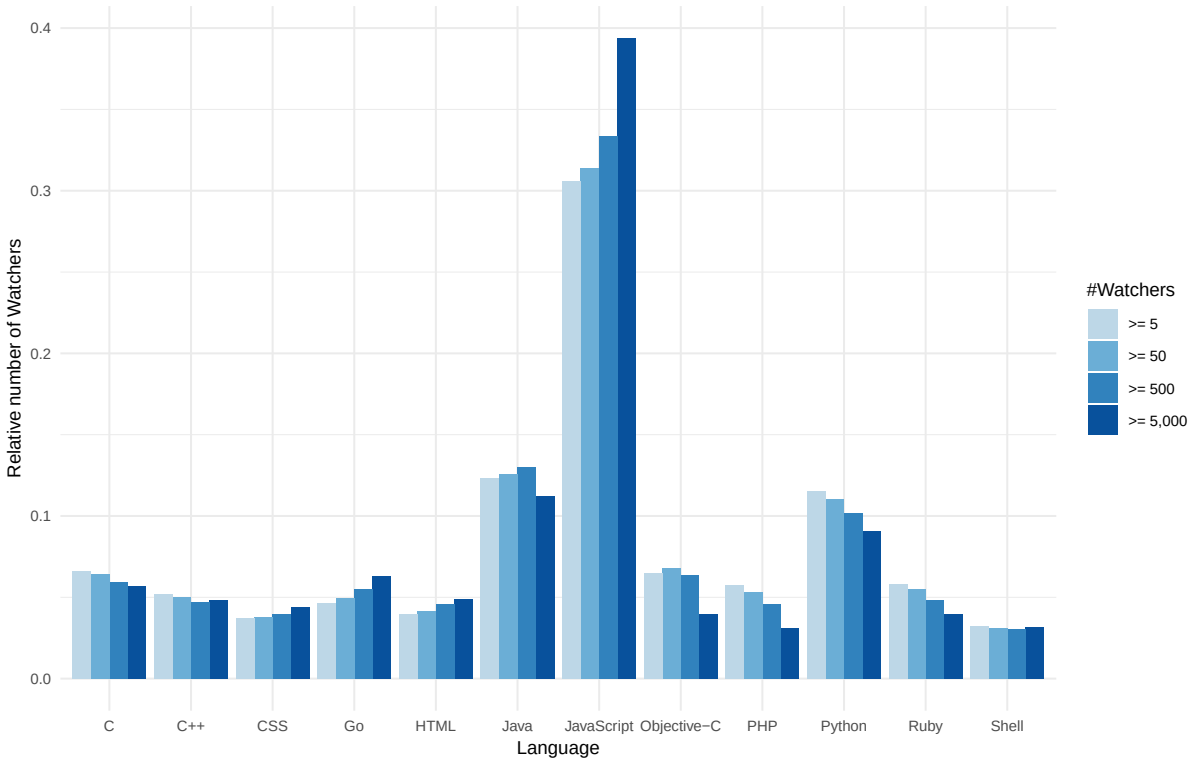
Table 10: Statistics on the sets of projects

Min. #Watchers	#Projects	Avg. #Watchers	Total #Watchers
5	1,112,989	80	85,776,331
50	199,261	366	73,119,359
500	25,212	1,897	48,878,405
5,000	1,617	11,346	18,297,552

Statistic values for the distribution of the number of projects, average number of watchers and total number of watchers, grouped by 4 different threshold values of watchers

7.3.3 (*) Summary

As already done in the motivational Section 2, we have analysed the popularity of programming languages. The key difference is that we make use of a graph database instead of a relational one. While we have not performed the same analyses to compare run-times between these models, SPARQL enabled us to write pattern-finding queries easily, allow-



Comparison for different implementations of a weighting by the number of watchers. Each grouping considers a different threshold value for the minimal number of required watchers. The popularity was measured by the relative number of watchers

Figure 18: Primary languages of projects weighted by watchers

ing us to explore more facets of the dataset. We have found that some countries focus more on web languages than others, and that geographical location seems to have an impact on programming language usage. As we considered a different aspect of popularity, moving from how developers use languages to how clients use resulting programs, we found huge differences between the popularity of languages. Unfortunately we were not able to implement one of the strongest advantages of RDF graphs: the ability to interlink databases. While we do have outgoing links to DBpedia, we were unable to load their dataset into our triplestore due to insufficient computational resources. Further trends could be explored, for example by using the `nearestCity` relation [12], or by comparing if users from large cities act different to users from the countryside, querying the population of a location. [13]

[12] http://dbpedia.org/ontology/nearestCity
[13] http://dbpedia.org/ontology/population

8 (*) Expanding and Linking the Data

In this section, we will describe how the dataset can be extended to include further `git` hosts or interlinked with external data. In Subsection 8.1, we give an example extension for GitLab [88], making use of the design choices from Section 4.1. In Section 8.2, we show how the dataset can be aligned with external data, where our focus will lie on interlinking our data with DBpedia [11].

8.1 (*) New Data Sources

As SemanGit explores semantics on the `git` protocol in general, it is important that we do not consider data about GitHub as only potential source of data. Unfortunately, the resources we have at hand are insufficient to expand our dataset further, as we are already facing many difficulties due to the size of our dataset. However, we will present an extension to our ontology to demonstrate the effectiveness of the chosen approach from Section 4.1 that allows a convenient modelling of further `git` hosts with their custom features.

As GitLab also implements the `git` protocol, the underlying structure of the data one can gather using their API has many common facets to the one from GitHub. As such, many of the new classes in our extension are able to inherit from existing classes, resulting in a common structure. On the other hand, GitLab has also added custom features.

Our extension does not cover all information that could potentially be extracted from GitLab. At this time, it is supposed to serve just as an example extension to demonstrate how common features can be inherited and custom features can be added separately. Due to a rate limit on the API [20], it is most likely infeasible to extract all information the API could potentially deliver. Currently, the extension holds 51 classes with 303 properties as of April 16, 2019 and can be found on our GitHub repository,[14] including a link to a visualisation with WebVOWL [6]. We will now proceed to mention some key points about the extension, which also led us to making a few changes to the prior ontology.

Fake users – The author of a `gitlab_commit` is not a `gitlab_user`, but is described by literals for name and email instead. The same holds true for the committer of a `gitlab_commit`. Note that committer and author of a commit can differ. This is in

[14] `https://github.com/SemanGit/SemanGit/blob/master/Documentation/ontology/gitlabextension.ttl`

accordance with the returned result of the GitLab API, not returning a user, but just a name and an email address. We attribute this behaviour to the fact that the author of a `gitlab_commit` does not need to be a registered GitLab user, as another person could potentially push this commit to the GitLab repository. A comparable behaviour can be observed when looking at the data extracted from GitHub by GHTorrent, where some users are marked as "fake users" which are described as follows [5]:

> *If the commit user has not been resolved, for example because a commit user is not a Github user or the git user's name is misconfigured, GHTorent will create a fakeuser entry with as much information as available.*

Commit statistics – The original ontology was not considering any statistics for the general `commit` class for the `git` protocol. This includes the number of additions and deletions contained in a commit. GitLab explicitly provides these values for commits. While the GHTorrent data does not contain any information regarding these statistics, it is in the nature of a commit to consist of additions and deletions. For example, an altered line results in the deletion of the old line and the addition of a new line.

Branches – The `git` protocol specifies branches and how they can be merged. Previously, we did not have a class to encompass information regarding branches, as GHTorrent does not provide branching information in the monthly dumps we use. While this information is contained in the daily dumps of GHTorrent, these dumps are incremental and the additional information contained in them makes them much larger. As we are already struggling with adding a SPARQL endpoint on the dataset, we chose to keep using the monthly dumps until further resources are available. There are other relations only contained in daily dumps, such as metadata about a project's milestones.

8.2 (*) Interlinking with other Datasets

One of the strengths of RDF graphs is that they can be linked with other datasets. For the SemanGit, we would like to perform two types of interlinkage:

- Linking different accounts of the same user on various `git` hosting platforms with an `owl:sameAs` relation
- Linking to external datasets, such as DBpedia, to be able to access additional information about well known objects

The first part seems to be almost impossible to automate, as we do not have any identifying information about the users. The old dumps of GHTorrent contained email addresses and what users entered for the "Real Name" field. Changes in data protection law, most notably the General Data Protection Regulation (GDPR), forced GHTorrent to stop distributing this sensitive information [89]. It is still possible to match accounts listing the same username, but this method is prone to errors. Checking additional information like the reported geolocation might help to improve this approach. Nevertheless, as the user decides what information is published, such data might be faulty or not present. To automate such a matching, we would therefore need to implement machine learning algorithms or restrict the matchmaking process to a conservative policy, only accepting a match if all information is provided and identical for both data sources.

On the other hand, the second part is quite straightforward in a few scenarios and we already implemented a straightforward approach for interlinking geoinformation with DBpedia. The "Location" field of a GitHub user is still available to us and GHTorrent even provides geocoded information about this location field, including a user's state and city. Obviously, not all users fill out the location field, or might enter false or even comical locations. In the case of real location entries, meaning non fictional location names, we are able to refer to the user's city and state on DBpedia by converting the location names to appropriate URIs. This could add additional information, if the DBpedia dataset could also be loaded into our triplestore. For example, we could query the population density of the city the user is living in, to compare behavioural traits of users from major cities with those from the countryside. Such interlinking could also prove to be particularly useful when using the SemanGit dataset for the scenario of headhunting, as one could not only search for users from a certain city, but also for users from surrounding villages or nearby cities, by making use of the "dbo:nearestCity" relation.

Lastly, besides linking to external datasets, we also re-use external ontologies. As an example, we are using the Vocabulary of Interlinked Datasets (VoID) to describe our dataset. This metadata includes information about the authors, how it is published, when it was last updated, and more. Re-using further datasets could improve the quality of our ontology.

9 (*) Towards an Industrial Deployment

Up to this section, all analyses we described were performed on the entire data without sampling. The trick we used was to only load the relations that our analyses require, skipping all others. Due to our limited computational resources, we could not investigate some facets with promising industrial use cases. The main hold up is caused by the single node load in process into our triplestore, which reads the RDF files and creates a journal file, see Section 6.2. To showcase the value of an industrial deployment of SemanGit, where additional resources are at hand, we have created some examples for analyses and queries, using a small `head` sample of our dataset to evaluate queries, see Section 5.2. In Subsection 9.1, we take a closer look at analysing information gathered from the issue tracking system of GitHub. In Subsection 9.2, we briefly discuss the concept of *Ghost Contributors*, elaborating how a company might be able to use this for enticing project managers away from other companies.

9.1 (*) Analysing Reported Issues

GitHub comes with its own issue tracking features. Users are able to submit issue reports to repositories, unless issue tracking is disabled for the project, to file bugs or enhancements. Collaborators can be assigned to deal with these issues and labels, such as *Bug, Enhancement, Question, Duplicate, Invalid, Wontfix, Help wanted.* Also custom tags, can be assigned to issues by collaborators of the repository. By default, there is also a label "good first issue" to encourage new reporters to keep up their good work. Besides information about the assigned labels, we also have meta data about the issue reporter, the assignee (if any) and timestamped actions (opened, closed, re-opened, corresponding pull request merged, assigned, user (un-)subscribed updates). Note that GHTorrent creates a generic issue for every pull request. This can be interpreted as an issue report including a suggestion for a fix. Furthermore, issues can be commented. The dataset includes details about the comment author, creation date and the corresponding issue. In the case of a pull request, further information, including the body text itself, is available. If all this information were loaded into a triplestore, a multitude of analyses could be performed, such as:

1. Which kind of projects encourage people to not only report issues, but also contribute to fixing them?
2. Which developers are fast to respond to reported or assigned issues, compared to others within an organisation?

3. Is there a positive correlation between listening to the community and creating a successful project?

These are just some sample analyses, showing the analytical potential contained within issues. *Coelho et al.* [60] managed to detect new, undocumented errors and bug sources in Android and proprietary libraries, by analysing 160,000 reported issues related to Android projects.

Many open source software projects are too large to be handled by a small number of collaborators. It could be beneficial to not only have users reporting issues related to the project, but also users contributing code actively via pull requests, to also help with fixing these issues. For any pull request, GHTorrent stores the `head repository` and `base repository`. One could therefore compare the number of *external* pull requests with the number of reported issues for a project. This would yield an indicator for how involved the community is. Those projects with an involved community might share certain features, such as programming language paradigms, quality of documentation or more subtle features.

Analysis (2) is less abstract and can be discussed more explicitly. For quality assurance purposes, it might be interesting for a company to know which developer is quick to react to reported issues and who tends to be slow. Except for speed, one could also look at the quality of the fixes, by analysing the number of times issues are re-opened. A basic SPARQL query, listing the fastest fixing developers within a given organisation, is illustrated in Query 6.

Query 6 is a rudimentary example to show how a SPARQL query that operates on issues could look like. Several more things need to be considered within a real-scenario query:

- The query compares the timestamp of when the issue was assigned to a developer, with the timestamp of the issue being closed. If the issue is re-opened and closed again, there will be two "issue closing events".
- Additional events should be taken into account, such as the developer making a comment for clarification.
- Issues can be re-assigned. This means that the original assignee should be excluded.

```
PREFIX sgo: <http://semangit.de/ontology/>
SELECT ?user (AVG(?time_taken) AS ?average_response_time)
{
 BIND(sgo:github_project\/7k4 AS ?project)
 ?issue sgo:github_issue_project\/ ?project .
 ?issue_event sgo:github_issue_event_for\/ ?issue ;
              sgo:github_issue_event_action\/ "assigned" ;
              sgo:github_issue_event_created_at\/ ?t1 .
 ?issue_event2 sgo:github_issue_event_for\/ ?issue ;
               sgo:github_issue_event_action\/ "closed" ;
               sgo:github_issue_event_created_at\/ ?t2 .
 ?issue sgo:github_issue_assignee\/ ?user .
 BIND((?t2 - ?t1) AS ?time_taken)
} GROUP BY ?user
```

Query 6: SPARQL query to list the developers of an organisation who are on average fastest to respond to issues.

Analysis (3) aims at respecting the suggestions made by the community. To do so, issue labels could be of particular interest. Do users who reported an issue and were awarded with the "good first issue" label tend to submit more suggestions, or is the quality of future suggestions of such a user affected positively by such an event? Is there an opposite effect for users whose issues are labelled with "wontfix"? Also, the response time, as discussed in Analysis (2) could be integrated into this analysis.

The results of performing these analyses could reveal new insights on how successful project management should be done on open source software platforms. While we did not conduct any of the analyses presented in this section, we believe that issue related analyses have great potential in an industrial deployment. Not only could a company perform these analyses for improving the internal workflow, but also to apply similar techniques for finding suitable developers on GitHub they could hire.

9.2 (†) Ghost Contributors

Often, the larger organisations get, the more hierarchy they develop for managing their work. With respect to companies on GitHub, this could for example mean that there are developers under the supervision of a project manager. The main responsibilities of the manager is to make sure that the project evolves smoothly, but she does not necessarily write code by herself. Finding such patterns could be valuable for many use cases, such as headhunting, where a company tries to find people with certain skills to hire. Uncovering the role of a user within an organisation could provide the opportunity for other

companies to recruit project managers with sufficient expertise.

In an attempt to find such managers, we will introduce the concept of **Ghost Contributors**. These are GitHub users, who are collaborators of a project, but do not contribute code to it. Obviously, there are other scenarios in which ghost contributors could occur other than project managers. For example, a student developing a program for an assignment, who "shares" the project with his tutor, i.e. invites him to collaborate. This tutor would also classify as ghost contributor. By showing the existence of such users, we also want to warn that using *number of contributors of a project* as weight for a project in an analysis, as done by GitHub in their language analysis [16], has some perils.

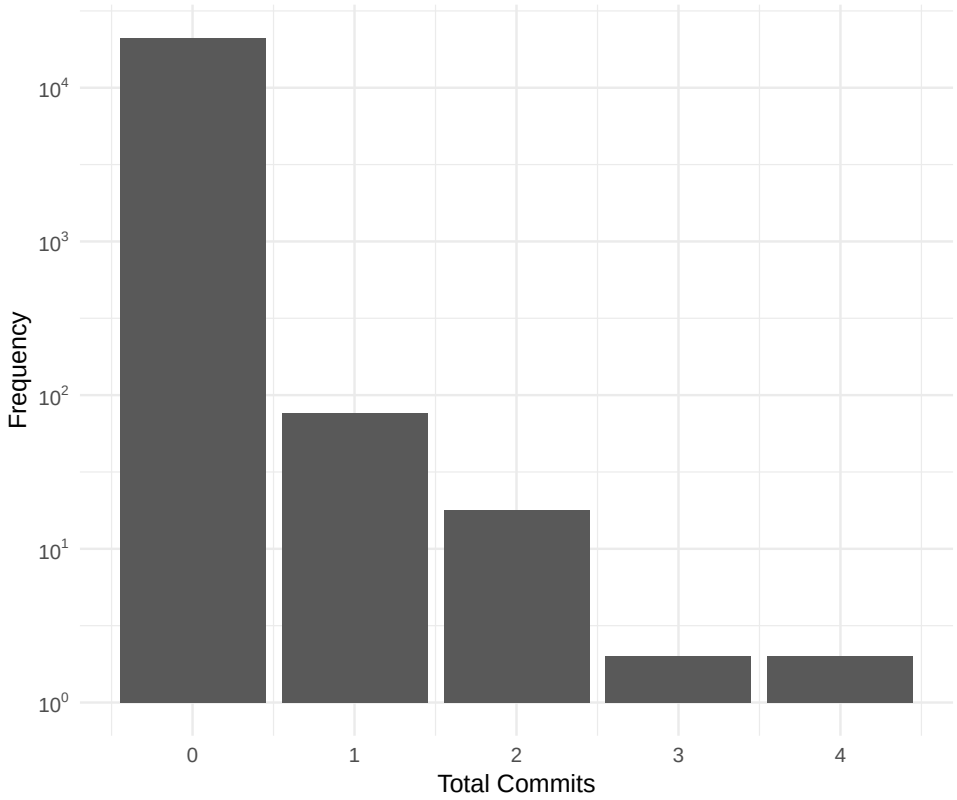
```
PREFIX sgo: <http://semangit.de/ontology/>
SELECT ?user ?org (COUNT(?commit) AS ?totalCommits) {
  {
    --Subquery to find only projects of relevant size
    SELECT ?project (COUNT(?commit) AS ?commitCtr)
           (COUNT(DISTINCT ?user) AS ?userCtr) {
        --project must be owned by an organisation
        ?org sgo:github_user_is_org\/ true .
        ?project sgo:github_has_owner\/ ?org .
        ?join_event sgo:github_organization_is_joined\/ ?org ;
                    sgo:github_organization_joined_by\/ ?user .
        ?commit sgo:commit_belongs_to_repository\/ ?project.
    } GROUP BY ?project HAVING((?commitCtr > 50) && (?userCtr > 2))
  }
  ?project sgo:github_has_owner\/ ?org .
  ?joinEvent sgo:github_organization_is_joined\/ ?org ;
             sgo:github_organization_joined_by\/ ?user ;
             sgo:github_organization_joined_at\/ ?t1 .
  --Only consider users who were a member for longer than 30 days
  FILTER((now() - ?t1) > 30)
  --Count commits. Optional is added to also capture those with 0 commits
  OPTIONAL {
    ?commit sgo:commit_belongs_to_repository\/ ?project ;
            sgo:commit_author\/ ?user .
  }
} GROUP BY ?org ?user
HAVING(?totalCommits < 5) ORDER BY ASC(?totalCommits)
```

Query 7: SPARQL query to find ghost contributors across organisations with at least 3 members and 50 commits.

In this section, we will focus on the task of finding users who are potentially in a management position. For this reason, we will add additional constraints to the ghost

contributors we find, such as a minimum organisation size of three members, which has a project with at least 50 commits. Realistically, these values should be chosen higher, but we are performing our queries on a sample in this section. Furthermore, to exclude newcomers within an organisation, we require a membership within the organisation of at least one month. The resulting SPARQL query is shown in Query 7.

This query is meant to prove the existence of ghost contributors within organisations. In an industrial deployment, further aspects should be considered within the query, such as how long a user has been part of an organisation. As an example, a user who pushed 20 commits within one month of membership, should not be considered a ghost contributor. In contrast, a user who pushed 20 commits over three years of membership probably should be. Additional behavioural patterns should also be taken into account to improve the precision of classification, such as whether or not the ghost contributor creates issues, comments on commits and accepts or rejects pull requests.



Plot of the number of users per organisation, with less than 5 commits. The data is extracted from the results of Query 7. The display uses a logarithmic scale.

Figure 19: Number of users by commits within organisations

Query 7 took 1h1m to compute and returned a total of 21,197 results, when executed on our sample of 687,586,749 triples. The majority of users returned by the query have no

commits at all within the organisations they reside in, see Figure 19. We can clearly see the existence of ghost contributors. However, further filtering will be required to classify them into subgroups, such as managers.

10 (*) Conclusion

In this thesis, we have introduced the SemanGit dataset, discussed how it can be loaded for query purposes, performed first analyses and provided suggestions for further analyses. In the motivational Section, a language analysis was performed, yielding first results on the data contained in SemanGit, while also learning about the suitability of different kinds of analyses. We moved on to describing how the SemanGit dataset is created, by giving details about the ontology and conversion process, so that other researchers could repeat our experiments. Due to issues that naturally arise with large graphs, we elaborate how samples can be created which are suitable for analyses on the SemanGit dataset. We found that loading the entire dataset is prohibitive with the resources available to us. Multiple experiments were carried out to reveal appropriate methods for loading relevant data into a triplestore. While we failed to load the entire dataset, we still managed to design analyses and queries and performed them on the full dataset, by only loading the sections of the dataset that the queries operate on. To demonstrate the flexibility of our ontology and the strengths of graph databases, we created a sample dataset extension to another `git` hoster and described how the dataset can be interlinked with DBpedia. Lastly, we elaborated how such a large scale dataset could be used in an industrial deployment, where additional resources are at hand.

SemanGit faces certain limitations. Despite the potential value of the dataset, the effort to establish a queryable endpoint is not to be underestimated. In our setup, where only a single commodity machine was available, the loading of the entire data is prohibitively expensive, not to mention interlinkage with further datasets or including data from more `git` repository providers. Therefore, if computational resources are similar to ours, analytical possibilities are limited. One can either decide to only load the data relevant for queries which were designed beforehand, see Section 7, or operate on a sufficiently small sample as demonstrated in Section 9. However, generating such a sample with analytical value is far from trivial, as we had to conclude in Section 5.

Future work with SemanGit includes a re-modelling of the ontology, to increase the re-usability of our work. We were unable to update the ontology towards the later stages of the thesis, as this would have invalidated all queries performed thus far, and re-loading the dataset with an updated ontology is prohibitive. Additional features can be added to the ontology, such as some inverse relations, and the "camel case" naming option could be implemented. One could strive to re-use existing vocabularies more. Furthermore, our

Java conversion tool, as described in Section 4.2, could be made more dynamic, possibly enough to re-use it to include additional data sources. Also, as one can see from our SPARQL queries, the converter places a slash "/" not only to separate the ID of an entity from the ontology part (e.g. `<http://semangit.de/ontology/github_project/123>`), but also at the end of relation and class names, resulting in unconventional query formats, such as `?p a sgo:github_project\/`. We will strive to process the latest GHTorrent data at least every two months and provide an updated dataset dump on our website.

Despite these limitations, we find that the dataset has enormous analytical potential. This is supported by the fact that a conference paper about SemanGit has been accepted by and will be presented at the International Semantic Web Conference (ISWC) 2019 [90], where we hope to meet many like minded researchers with interest in a semantic dataset about `git`. Besides the insights we have gained from our example analyses, most extensive of which has been the investigation of various aspects about programming languages, we can already provide this dataset to the public, so that it can be used in upcoming open source software research, of which there has been a vast amount already [4]. Instead of focusing on any analysis in particular, we attempted to showcase the wide range of use cases for the data. We conclude that the dataset is valuable both for industrial usage, as well as for other researchers, such as those from the open source software area.

Further Reading

Website: `http://semangit.de/`
Lab Project Report: `http://semangit.de/project_report.pdf`
GitHub Repository: `https://github.com/SemanGit/SemanGit/`
SemanGit: A Linked Dataset from `git`: To be published in ISWC 2019 [90]
Ontology: In our GitHub repository. Visualisation on our website.

References

[1] Git. https://git-scm.com/. Accessed: March 11, 2019.

[2] Comparison of source-code-hosting facilities. https://en.wikipedia.org/wiki/Comparison_of_source-code-hosting_facilities. Accessed: March 6, 2019.

[3] GitHub About. https://github.com/about. Accessed: June 5, 2019.

[4] V. Cosentino, J. L. Cánovas Izquierdo, and J. Cabot. A systematic mapping study of software development with github. *IEEE Access*, 5:7173–7192, 2017.

[5] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[6] Steffen Lohmann, Vincent Link, Eduard Marbach, and Stefan Negru. Webvowl: Web-based visualization of ontologies. In Patrick Lambrix, Eero Hyvönen, Eva Blomqvist, Valentina Presutti, Guilin Qi, Uli Sattler, Ying Ding, and Chiara Ghidini, editors, *Knowledge Engineering and Knowledge Management*, pages 154–158, Cham, 2015. Springer International Publishing.

[7] OWL 2 Web Ontology Language Primer (Second Edition). https://www.w3.org/TR/owl2-primer/. Accessed: March 11, 2019.

[8] RDF Schema 1.1. https://www.w3.org/TR/rdf-schema/. Accessed: March 11, 2019.

[9] RDF Primer. https://www.w3.org/TR/rdf-primer/. Accessed: March 11, 2019.

[10] W3C Semantic Web Frequently Asked Questions. https://www.w3.org/RDF/FAQ.html. Accessed: June 12, 2019.

[11] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[12] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.

[13] SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/. Accessed: May 6, 2019.

[14] LOD Cloud: DBpedia dataset description. https://lod-cloud.net/dataset/dbpedia. Accessed: June 12, 2019.

[15] GitHut - Programming Languages and GitHub. https://githut.info/. Accessed: May 6, 2019.

[16] The State of Octoverse. https://octoverse.github.com/projects.html#languages. Accessed: March 5, 2019.

[17] GHTorrent Database Schema Description. http://ghtorrent.org/relational.html. Accessed: March 5, 2019.

[18] GitHub Pages. https://pages.github.com/. Accessed: March 5, 2019.

[19] Understanding rate limits for GitHub Apps. https://developer.github.com/apps/building-github-apps/understanding-rate-limits-for-github-apps/. Accessed: March 6, 2019.

[20] Implement rate limiting for the API endpoint. https://gitlab.com/gitlab-cookbooks/gitlab-haproxy/merge_requests/49. Accessed: April 16, 2019.

[21] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, Oct 2016.

[22] J. Howison and K. Crowston. The perils and pitfalls of mining sourceforge. *IET Conference Proceedings*, pages 7–11(4), January 2004.

[23] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34, December 2015.

[24] Gregorio Robles and Jesus M. Gonzalez-Barahona. Developer identification methods for integrated data from various sources. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[25] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global, 2011.

[26] W3 LargeTripleStores. `https://www.w3.org/wiki/index.php?title=LargeTripleStores&action=history`. Accessed: March 21, 2019.

[27] Bill Beauregard. Oracle Spatial and Graph: Benchmarking a Trillion Edges RDF Graph. `https://download.oracle.com/otndocs/tech/semantic_web/pdf/OracleSpatialGraph_RDFgraph_1_trillion_Benchmark.pdf)`, 2016.

[28] Campbridge Semantics Inc. TRILLION TRIPLES BENCHMARKING. `https://info.cambridgesemantics.com/cambridge-semantics-shatters-previous-record-of-loading-and-querying-trillion-triples`-by-100x.

[29] Franz Inc. AllegroGraph RDFStore Benchmark Results. `https://franz.com/agraph/allegrograph/agraph_benchmarks.lhtml)`.

[30] Francesco Corcoglioniti, Marco Rospocher, Michele Mostarda, and Marco Amadori. Processing billions of rdf triples on a single machine using streaming and sorting. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 368–375, New York, NY, USA, 2015. ACM.

[31] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. Query-oriented summarization of rdf graphs. *Proceedings of the VLDB Endowment*, 8(12):2012–2015, 2015.

[32] Laurens Rietveld, Rinke Hoekstra, Stefan Schlobach, and Christophe Guéret. Structural properties as proxy for semantic relevance in rdf graph sampling. In *International Semantic Web Conference*, pages 81–96. Springer, 2014.

[33] Christian Hübler, Hans-Peter Kriegel, Karsten Borgwardt, and Zoubin Ghahramani. Metropolis algorithms for representative subgraph sampling. In *2008 Eighth IEEE International Conference on Data Mining*, pages 283–292. IEEE, 2008.

[34] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 04 1970.

[35] Xuesong Lu and Stéphane Bressan. Sampling connected induced subgraphs uniformly at random. In *International Conference on Scientific and Statistical Database Management*, pages 195–212. Springer, 2012.

[36] RDF Stream Processors Implementation. https://www.w3.org/community/rsp/wiki/RDF_Stream_Processors_Implementation. Accessed: Jun 16, 2019.

[37] Amadou Fall Dia, Zakia Kazi-Aoul, Aliou Boly, and Yousra Chabchoub. C-sparql extension for sampling rdf graphs streams. In *Advances in Knowledge Discovery and Management*, pages 23–40. Springer, 2018.

[38] streamreasoning GitHub Page. https://github.com/streamreasoning/. Accessed: June 17, 2019.

[39] Josh Lerner and Jean Tirole. Some simple economics of open source. *The Journal of Industrial Economics*, 50(2):197–234, 2002.

[40] Tracii Ryan and Sophia Xenos. Who uses facebook? an investigation into the relationship between the big five, shyness, narcissism, loneliness, and facebook usage. *Computers in human behavior*, 27(5):1658–1664, 2011.

[41] Namsu Park, Kerk F Kee, and Sebastián Valenzuela. Being immersed in social networking environment: Facebook groups, uses and gratifications, and social outcomes. *CyberPsychology & Behavior*, 12(6):729–733, 2009.

[42] Yoram Bachrach, Michal Kosinski, Thore Graepel, Pushmeet Kohli, and David Stillwell. Personality and patterns of facebook usage. In *Proceedings of the 4th annual ACM web science conference*, pages 24–32. ACM, 2012.

[43] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. Information network or social network?: the structure of the twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 493–498. ACM, 2014.

[44] F. Thung, T. F. Bissyandé, D. Lo, and L. Jiang. Network structure of social coding in github. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 323–326, March 2013.

[45] Apache Cassandra. https://developer.github.com/v3/activity/starring/. Accessed: June 8, 2019.

[46] Kelly Blincoe, Jyoti Sheoran, Sean Goggins, Eva Petakovic, and Daniela Damian. Understanding the popular users: Following, affiliation influence and leadership on github. *Information and Software Technology*, 70:30 – 39, 2016.

[47] Yue Yu, Gang Yin, Huaimin Wang, and Tao Wang. Exploring the patterns of social behavior in github. In *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies*, CrowdSoft 2014, pages 31–36, New York, NY, USA, 2014. ACM.

[48] Bogdan Vasilescu. Human aspects, gamification, and social media in collaborative software engineering. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 646–649, New York, NY, USA, 2014. ACM.

[49] Jwen Fai Low, Tennom Yathog, and Davor Svetinovic. Software analytics study of open-source system survivability through social contagion. In *2015 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 1213–1217. IEEE, 2015.

[50] Lingxiao Zhang, Yanzhen Zou, Bing Xie, and Zixiao Zhu. Recommending relevant projects via user behaviour: an exploratory study on github. In *CrowdSoft@SIGSOFT FSE*, 2014.

[51] Sagar Sachdeva. Automated Social Recruiting through GitHub. Master's thesis, University of Dublin, Trinity College, 2017.

[52] Fernando Figueira Filho, Marcelo Gattermann Perin, Christoph Treude, Sabrina Marczak, Leandro Melo, Igor Marques da Silva, and Lucas Bibiano dos Santos. A study on the geographical distribution of brazil's prestigious software developers. *Journal of Internet Services and Applications*, 6(1):17, Aug 2015.

[53] H. Lee, B. Seo, and E. Seo. A git source repository analysis tool based on a novel branch-oriented approach. In *2013 International Conference on Information Science and Applications (ICISA)*, pages 1–4, June 2013.

[54] Ayushi Rastogi, Nachiappan Nagappan, Georgios Gousios, and André van der Hoek. Relationship between geographical location and evaluation of developer contributions in github. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, pages 22:1–22:8, New York, NY, USA, 2018. ACM.

[55] Josh Terrell, Andrew Kofink, Justin Middleton, Clarissa Rainear, Emerson Murphy-Hill, Chris Parnin, and Jon Stallings. Gender differences and bias in open source:

pull request acceptance of women versus men. *PeerJ Computer Science*, 3:e111, May 2017.

[56] Yuri Takhteyev and Andrew Hilts. Investigating the geography of open source software through github. *Manuscript submitted for publication*, 2010.

[57] David Rusk and Yvonne Coady. Location-based analysis of developers and technologies on github. In *Proceedings of the 2014 28th International Conference on Advanced Information Networking and Applications Workshops*, WAINA '14, pages 681–685, Washington, DC, USA, 2014. IEEE Computer Society.

[58] Dorota Celińska. Coding together in a social network: Collaboration among github users. In *Proceedings of the 9th International Conference on Social Media and Society*, SMSociety '18, pages 31–40, New York, NY, USA, 2018. ACM.

[59] Ali Sajedi Badashian and Eleni Stroulia. Measuring user influence in github: The million follower fallacy. In *Proceedings of the 3rd International Workshop on Crowd-Sourcing in Software Engineering*, CSI-SE '16, pages 15–21, New York, NY, USA, 2016. ACM.

[60] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. Unveiling exception handling bug hazards in android based on github and google code issues. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 134–145. IEEE, 2015.

[61] SemanGit: Adding a semantic layer to the Git protocol. `https://github.com/SemanGit/SemanGit/blob/master/Documentation/project_report.pdf`.

[62] Alexa Traffic Rank for GitHub. `https://www.alexa.com/siteinfo/github.com`. Accessed: May 23, 2019.

[63] Alexa Traffic Rank for SourceForge. `https://www.alexa.com/siteinfo/sourceforge.net`. Accessed: May 23, 2019.

[64] Open Semantic Framework: Ontology Best Practices. `http://wiki.opensemanticframework.org/index.php/Ontology_Best_Practices`. Accessed: June 20, 2019.

[65] RDF 1.1 Turtle. `https://www.w3.org/TR/turtle/`. Accessed: March 6, 2019.

[66] RDF 1.1 N-Triples. `https://www.w3.org/TR/2014/REC-n-triples-20140225/`. Accessed: June 19, 2019.

[67] Apache Cassandra. http://cassandra.apache.org/. Accessed: May 16, 2019.

[68] Linux kernel source tree. https://github.com/torvalds/linux. Accessed: May 17, 2019.

[69] Oracle Spatial and Graph. https://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.html. Accessed: March 20, 2019.

[70] Stardog. https://www.stardog.com/. Accessed: March 20, 2019.

[71] AllegroGraph. https://franz.com/agraph/allegrograph/. Accessed: March 20, 2019.

[72] OpenLink Virtuoso. https://virtuoso.openlinksw.com/. Accessed: March 20, 2019.

[73] The State of Octoverse. http://graphdb.ontotext.com/. Accessed: March 20, 2019.

[74] Blazegraph. https://www.blazegraph.com/. Accessed: March 20, 2019.

[75] Felix Conrads, Jens Lehmann, Muhammad Saleem, Mohamed Morsey, and Axel-Cyrille Ngonga Ngomo. I guana: a generic framework for benchmarking the read-write performance of triple stores. In *International Semantic Web Conference*, pages 48–65. Springer, 2017.

[76] Dieter De Witte, Laurens De Vocht, Ruben Verborgh, Kenny Knecht, Filip Pattyn, Hans Constandt, Erik Mannens, and Rik Van de Walle. Big linked data etl benchmark on cloud commodity hardware. In *Proceedings of the International Workshop on Semantic Big Data*, SBD '16, pages 12:1–12:6, New York, NY, USA, 2016. ACM.

[77] Ghislain Auguste Atemezing and Florence Amardeilh. Benchmarking commercial rdf stores with publications office dataset. In *European Semantic Web Conference*, pages 379–394. Springer, 2018.

[78] Vassilis Papakonstantinou, Claus Stadler, Michael Röder, and Axel-Cyrille Ngonga Ngomo. Mocha2018: The mighty storage challenge at eswc 2018. *Semantic Web Challenges: 5th SemWebEval Challenge at ESWC 2018, Heraklion, Greece, June 3-7, 2018, Revised Selected Papers*, 927:3, 2018.

[79] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. Reifying rdf: What works well with wikidata? In *11th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2015)*, page 32, 2015.

[80] Revision history of "LargeTripleStores". `https://www.w3.org/wiki/index.php?title=LargeTripleStores&action=history`. Accessed: March 6, 2019.

[81] Blazegraph Hardware Configuration. `https://wiki.blazegraph.com/wiki/index.php/Hardware_Configuration`. Accessed: March 11, 2019.

[82] Blazegraph Configuration. `https://wiki.blazegraph.com/wiki/index.php/Configuring_Blazegraph`. Accessed: March 21, 2019.

[83] Revision history of "LargeTripleStores". `https://wiki.blazegraph.com/wiki/index.php/Configuring_Blazegraph`. Accessed: January 24, 2019.

[84] Blazegraph IOOptimization. `https://wiki.blazegraph.com/wiki/index.php/IOOptimization`. Accessed: March 11, 2019.

[85] Timeline of GitHub. `https://en.wikipedia.org/wiki/Timeline_of_GitHub`. Accessed: May 6, 2019.

[86] Programming Languages — Ruby. IPA Ruby Standardization WG Draft. `https://www.ipa.go.jp/files/000011432.pdf`. Accessed: June 26, 2019.

[87] Java (programming language) - Wikipedia. `https://en.wikipedia.org/wiki/Java_(programming_language)`. Accessed: May 9, 2019.

[88] The first single application for the entire DevOps lifecycle - GitLab. `https://about.gitlab.com/`. Accessed: April 4, 2019.

[89] GHTorrent - Access to personal data. `http://ghtorrent.org/pers-data.html`. Accessed: April 18, 2019.

[90] Dennis O. Kubitza, Matthias Böckmann and Damien Graux. SemanGit: A Linked Dataset from git. In *to appear in the Proceedings of the 18th International Semantic Web Conference*, 2019.