Empirical Study on the Usage of Graph Query Languages in Open Source Java Projects

Philipp Seifer
University of Koblenz-Landau
Software Languages Team
Koblenz, Germany
pseifer@uni-koblenz.de

Johannes Härtel
University of Koblenz-Landau
Software Languages Team
Koblenz, Germany
johanneshaertel@uni-koblenz.de

Martin Leinberger
University of Koblenz-Landau
Institute WeST
Koblenz, Germany
mleinberger@uni-koblenz.de

Ralf Lämmel
University of Koblenz-Landau
Software Languages Team
Koblenz, Germany
laemmel@uni-koblenz.de

Steffen Staab
University of Koblenz-Landau
Koblenz, Germany
University of Southampton
Southampton, United Kingdom
staab@uni-koblenz.de

Abstract

Graph data models are interesting in various domains, in part because of the intuitiveness and flexibility they offer compared to relational models. Specialized query languages, such as Cypher for property graphs or SPARQL for RDF, facilitate their use. In this paper, we present an empirical study on the usage of graph-based query languages in open-source Java projects on GitHub. We investigate the usage of SPARQL, Cypher, Gremlin and GraphQL in terms of popularity and their development over time. We select repositories based on dependencies related to these technologies and employ various popularity and source-code based filters and ranking features for a targeted selection of projects. For the concrete languages SPARQL and Cypher, we analyze the activity of repositories over time. For SPARQL, we investigate common application domains, query use and existence of ontological data modeling in applications that query for concrete instance data. Our results show, that the usage of graph query languages in open-source projects increased over the last years, with SPARQL and Cypher being by far the most popular. SPARQL projects are more active in terms of query related artifact changes and unique developers involved, but Cypher is catching up. Relatively few applications use SPARQL to query for concrete instance data: A majority of those applications employ multiple different ontologies, including project and domain specific ones. Common application domains are management systems and data visualization tools.

CCS Concepts • **General and reference → Empirical studies**; • **Information systems → Query languages**; • **Software and its engineering → Software libraries and repositories**.

Keywords Empirical Study, GitHub, Graphs, Query Languages, SPARQL, Cypher, Gremlin, GraphQL

ACM Reference Format:
Philipp Seifer, Johannes Härtel, Martin Leinberger, Ralf Lämmel, and Steffen Staab. 2019. Empirical Study on the Usage of Graph Query Languages in Open Source Java Projects. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3357766.3359541

SLE '19, October 20–22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6981-7/19/10…$15.00
https://doi.org/10.1145/3357766.3359541

1 Introduction

Graph data models are interesting in various domains, in part due to the flexibility and intuitiveness they offer. Unlike the relational data model, schemata may be incrementally developed and adapted, or even omitted entirely. The graph model is also intuitive for modeling tasks, for example in general knowledge as exemplified by the Wikidata Knowledge Graph [70]. Specialized query languages facilitate the use of the graph data model. A long-standing standard is SPARQL [57], a query language for RDF graphs which conceptually matches on the triple structure of RDF. For property graphs, languages like Cypher [27] emerged, featuring specialized constructs to build results by selecting nodes and edges, either of which may potentially hold properties. While the major graph-related query approaches have been compared in terms of various properties of both language [4] and implementation [2, 34], we are interested in the practical

usage of such technologies. The focus of this study, therefore, is the comparison of overall popularity and practical use of graph-related query languages in real-world projects. More specifically, we analyze open-source Java projects on GitHub. Our primary *contributions* follow.

1. We compare the overall popularity of graph-related query languages in open-source projects, in terms of numbers of projects using the query languages, numbers of affected methods and files, numbers of developers involved and the changes of these numbers over time.
2. We use a refined, usage criteria-based approach to analyze usage of query languages in open-source projects, in terms of distinction of framework-like functionality versus concrete query usage and in terms of determining usage of ontologies.

The remainder of this paper is structured as follows. Section 2 motivates the selection of query languages we analyze and gives a short overview of these technologies. In Section 3, we introduce our research questions in depth and design the methodology of our study, to then present our results in Section 4. We conclude with related work in Section 5, before our summary and future work in Section 6.

2 Graph Query Languages

We restrict our study to popular graph-related query languages. Our selection is primarily based on two factors. First, we consider languages popular enough to occur in the GitHub Linguist classification [29]. For all 461 candidate languages[1] we manually identify query languages by considering resources such as project websites, GitHub repositories, or otherwise related Wikipedia entries. We classify the following 11 languages as query related: Flux, GraphQL, HiveQL, JsonIQ, Lasso, PLSQL, SPARQL, SQL, SQLPL, TSQL and XQuery. Here SPARQL and GraphQL [21, 30] are primarily related to graphs and therefore included in our comparison. As a baseline we also refer to XQuery [59] (an XML database query language) and SQL.

We suspect not all interesting query languages to be listed in Linguist, especially since not all languages must produce dedicated artifacts. Therefore, we use our identified languages as seed languages to find related work on graph-related query languages. Such work (e.g., [4, 34]) frequently refers to both Cypher [27, 52], the property graph query language initially developed for Neo4j [51], and the Gremlin graph traversal language [25, 60]. We do not include languages exclusive to proprietary graph databases, or more recent developments in research, such as G-CORE [3]. In both cases we do not expect representative results in open-source projects. Our study therefore focuses on the following four graph-related query languages. Examples are given in Figure 1, in the context of a simple graph consisting of *person* nodes and *knows* edges.

```
SELECT ?p ?k WHERE {
  ?p a foaf:Person ; foaf:knows ?k .
  ?k a foaf:Person
}
```

```
MATCH (p:Person)-[:KNOWS]->(k:Person)
RETURN p, k
```

```
g.V()
  .hasLabel('person').as('p')
  .outE('knows').inV()
  .hasLabel('person').as('k')
  .select('p', 'k')
```

```
person {
  name
  knows { name }
}
```

Figure 1. Example queries selecting people and the people they know in SPARQL (1st), Cypher (2nd), Gremlin (3rd) and GraphQL (4th).

SPARQL is a W3C standardized query language for RDF graphs, which at its core matches the triple structure (subject, predicate and object) of RDF data. SPARQL is strongly associated with the semantic web and commonly used conjointly with ontologies.

Cypher is a declarative query language for property graphs, initially developed for the Neo4j platform. Since 2015, the language is standardized by the openCypher [53] project. Cypher queries rely on matching graph patterns expressed with ASCII-Art syntax.

Gremlin is a graph traversal language with a focus on path-like expressions. It is the core graph matching language of the Apache Tinkerpop [24] framework.

GraphQL Unlike the other three query languages, the primary focus of GraphQL is not purely database access. Instead, GraphQL was developed as a graph-based replacement for the REST paradigm. In GraphQL, queries consist of JSON-like templates that are matched on server side schema definitions, usually in the context of web APIs.

[1] For the 519 languages that occur in this list, we consider all languages classified as `type: data` (95, e.g., SQL) or `type: programming` (366, e.g., C), while excluding those with `type` classifications of `markup` (45, e.g., HTML) and `prose` (13, e.g., Markdown). We include programming languages since some languages such as XQuery are not classified as data languages.

3 Methodology

Our empirical study targets open-source projects, with a scope restricted to Java projects and the GitHub platform. This section defines the two research questions we answer, as well as the design of our study in terms of data extraction and analysis. With our first research question we aim at understanding popularity and usage of the selected graph-based query languages.

RQ 1: *What is the overall popularity of different graph query languages in open-source Java projects and the development of query-related repository characteristics over time?*

We compare the aforementioned languages in terms of the following two aspects. We investigate the overall popularity (over time) of the respective query languages via the number of associated projects, while also comparing them to the number of SQL and XQuery projects as baselines. In addition, we analyze popularity in terms of GitHub stars.

For the most popular identified languages, we find projects that include queries in their source code. We compare the number of projects with and without queries. For projects that include queries, we analyze commit activity related to query containing methods and dedicated query files, as well as activity of the repositories in general. We also compare the number of contributing developers over time.

RQ 2: *What are characteristics of applications, selected via usage criteria, that employ concrete queries?*

With this second research question we refine our analysis via concrete usage criteria, by making a distinction between framework-like functionality and concrete query usage in applications. We focus our analysis on the latter and perform it for SPARQL, the most popular language as determined in the context of RQ1. For a selection of projects working with concrete queries, we find common application domains and investigate the usage of query kinds and data conceptualization in the form of ontologies.

Our data set collection, filtering and evaluation process is incremental and aligned with these two research questions. The remainder of this section is structured accordingly: We first describe the data collection methods for our initial data set of repositories, which forms the basis for all subsequent evaluation. We then introduce the commit history analysis and usage-criteria based filtering and ranking approaches we employ. Figure 2 outlines the steps of our general approach that are covered in Sections 3.1 through 3.5.

3.1 Dependency Selection

We select projects pertaining to the query languages introduced in Section 2 via related dependencies. In order to obtain such dependencies, we use the search function of mvnrepository [49], which indexes more than 14 million artifacts across various Maven repositories, including Maven Central [26]. We perform one search per technology, using the keywords `sparql`, `cypher`, `graphql` and `gremlin` respectively. The search considers metadata, including names and descriptions. See also Step 3.1 in Figure 2.

Table 1. Number of dependencies per query language, number of distinct groups (i.e., project names) and average number of dependencies per group (where ∼ marks estimations).

Language	Dependencies	Groups	Per Group
SPARQL	191	72	2.65
Cypher	80	14	**5.71**
GraphQL	**343**	140	2.45
Gremlin	84	35	2.40
XQuery	24	18	1.33
SQL	3336	*∼2058*	*∼1.62*

Each result we obtain consists of the group ID (the project name, for example `org.apache.jena`) and the artifact ID (jar name, for example `jena-arq`) for the respective dependency. The number of results for each language is listed in Table 1. Additionally, the table shows the number of unique groups and average number of artifacts per group. Overall, we find the most dependencies for GraphQL, followed by SPARQL. Cypher, while more recently becoming an open language standard, is still closely associated with Neo4j. Therefore, it occurs rarely outside the context of a few projects (namely Neo4j and openCypher) resulting in the high per-group average. For our baseline comparison, we also search for `xquery`, where we find 63 dependencies, and `sql` with 3336 dependencies. Due to limitations of the mvnrepository search, we can only retrieve up to 500 results. Therefore, groups and dependencies per group are estimations on a sample of 500 for SQL.

These initial results consist primarily of specific artifacts, such as the ARQ query processor of the Jena framework. However, common libraries provide convenient meta packages that include frequently used components. To accommodate for this, we extend the set of dependencies by clustering artifact names of the same group by their longest common prefix of at least length 4. With this approach we add a total of 18 SPARQL, 8 Cypher, 50 GraphQL and 11 Gremlin generalized dependency fragments to the dependency set.

3.2 Project Search

As a next step, we use the query language related dependencies obtained in Step 3.1 to search for projects on GitHub that depend on them. To this end, we consider the popular Java build systems Maven, Gradle and Ivy by performing code searches in files related to these systems. We query via the GitHub code search API for all combinations of our search terms (that is, occurrence of both the group ID and artifact

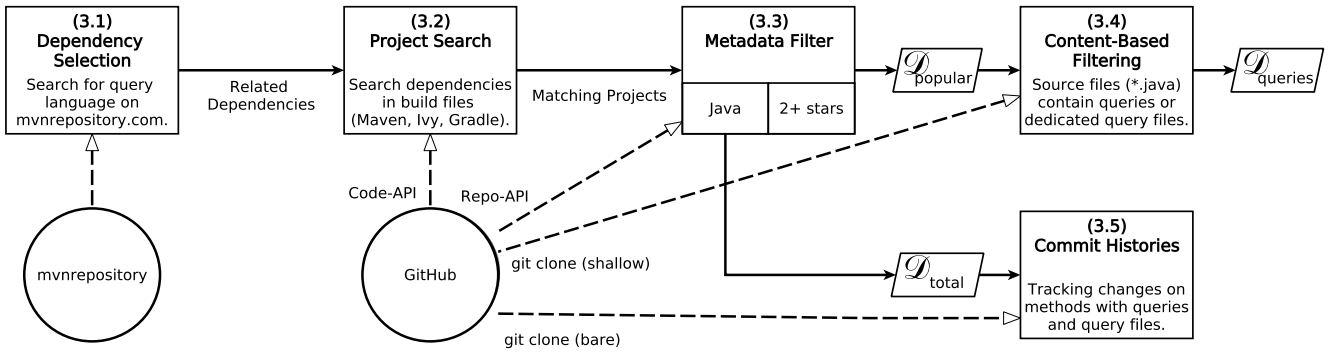

Figure 2. Overview of our general approach and resulting data sets $\mathcal{D}$ related to RQ1 and RQ2. Dashed lines represent access to external resources such as GitHub or mvnrepository. Solid lines represent data flow between filters.

Table 2. Build systems and associated queries per dependency consisting of a group ID `$g` and artifact ID `$a`.

System	Search Query
Maven	`/code?q=$g+$a+in:file+filename:pom.xml`
Gradle	`/code?q=$g+$a+in:file+extension:gradle`
Ant/Ivy	`/code?q=$g+$a+in:file+filename:ivy.xml`
Ant/Ivy	`/code?q=$g+$a+in:file+filename:build.xml`

ID) and build files. Table 2 lists the resulting four queries we execute per dependency. The table omits the common prefix `https://api.github.com/search` for brevity. From the resulting set of matching build files, we extract the sets of repositories for each of our query languages (i.e., repositories for which at least one build file was found, containing one or more dependencies related to the respective language).

Due to restrictions[2] of the GitHub API, only files on the default branch and smaller than 384 KB are considered. Since build files are commonly small, this should not impact results.

3.3 Metadata Filter

We obtain the repository metadata via the GitHub repository API for each related repository. We filter the initial results based on the following criteria, as is displayed in Step 3.3 of Figure 2.

Usage of Java While we target Java projects in our initial search via Java build systems such as Maven or Gradle, these build systems are also used for other languages. We therefore remove all repositories from our data set, that do not include Java in the set of languages obtained via the `languages_url` resource of the respective repository. We refer to this set of repositories as $\mathcal{D}_{total}$.

[2]We also avoid restrictions in the number of results returned by the code search API by segmentation of files based on a shifting window of file sizes.

Popularity For some of the following steps, we only consider repositories above a popularity threshold, using the number of stars (the `stargazers_count` property). Given the relatively small number of overall results, we consider repositories with at least 2 stars. We refer to this refinement of $\mathcal{D}_{total}$ as $\mathcal{D}_{popular}$.

3.4 Content-Based Filtering

We analyze repositories of the two most popular languages SPARQL and Cypher based on the occurrence of queries in Java source code and in dedicated query files (Step 3.4, also in Step 3.5). We consider queries that return values, which includes SELECT, ASK, DESCRIBE and CONSTRUCT queries in SPARQL and queries that use RETURN in Cypher. We refer to the data set of projects with queries as $\mathcal{D}_{queries}$. The following filter is used to determine whether a project contains queries in its source code.

For both languages, queries in source code are commonly represented as strings. Since Java has no direct support for multiple line strings, projects may use various well-known composition techniques: Query construction may range from `java.lang.string` methods, such as `join`, `format` or the `+` operator, to `StringBuilder` or Streams. For all approaches, the concatenation of string literals on a per-file basis yields query fragments, which may be incomplete if variables are included. To accommodate for this, we only match on the coarse structure of queries. In the case of SPARQL, we match query headers which are less likely to be affected by composition techniques or argument splicing. Our regular expressions are directly modeled after the SPARQL grammar [57]. For Cypher, we match on query headers such as `MATCH (...)` concluded by `RETURN`. As for SPARQL, we do not consider the potentially incomplete bodies of queries.

We validate this approach by sampling 10 random source files with queries per language (using simple search terms and manual filtering) and count the number of queries by hand, to a total of over 200. We also include additional 180 files without queries, false positives of the simple search and

files with the opposite query language. Our extraction has an accuracy of 97.4% and 100.0% for SPARQL and Cypher respectively (with no false positives in either case).

In addition to considering Java source files, we also include dedicated query files. For SPARQL, these are files with the commonly used extensions `.rq` or `.sparql` which are also assigned to SPARQL by GitHub Linguist. The `.rq` extension in particular is used by Jena ARQ [23]. For Cypher we consider the file extensions `.cyp` and `.cypher`.

3.5 Commit Histories

As the analysis of query related activity over the full lifespan of a repository is expensive, we sample from the Java repositories ($\mathcal{D}_{total}$) that are either related to Cypher or SPARQL. The analysis applied to this sample continuously tracks the activities on files and methods containing queries. To decide if an artifact contains queries, we use the file extension filters and query detection approaches as introduced in Step 3.4.

Activity at a certain point in time is measured in terms of changes done by a commit. First, the artifact content is converted into a bag of words to exclude simple effects of reordering within content. Changes are extracted by comparing the bag representation of a method or file before and after a commit. Added and removed terms are interpreted as changes. We track developers and artifacts involved in a query activity by attributing such change to the respective developer doing the commit, to the changed artifact's identification (a path or qualified method name) and to the corresponding timestamp. The resulting data can be used to compute the number of distinct repositories, developers and artifacts involved in a query activity at one time, over a timespan or in a window.

At the technical level, we perform parsing to extract method content; we track simple refactorings of files and methods to prevent wrong changes according to [5]; we consider branches to correctly capture changes that are merged into the master [42]; and we determine characteristics of SPARQL and Cypher repositories by performing a linear regression analysis to generalize on the population that we sampled from.

3.6 Content-Based Ranking

For our first research question, we select projects that are in any way related to the target languages. Such projects may range from framework extensions to databases and triple stores, or even approaches that employ SPARQL for its reasoning capabilities.

For our second research question, we aim to identify applications that make use of concrete queries, by refining our initial selection of projects. We perform this in-depth analysis for the most actively used language SPARQL. To this end, we first define the following categories of possible use cases for the SPARQL query language in various types of projects. These insights were gained from manual exploration of our preliminary results. In particular, SPARQL may be part of a projects source code in the following scenarios:

- **C1**. **Query Framework**. Libraries, frameworks and framework extensions that interface between concrete applications (via API or DSL) and back-end components, such as triple stores. Since such libraries facilitate the use of SPARQL, queries may be a part of either usage examples or test cases.
- **C2**. **Query Processor or Executor**. Applications that process queries (e.g., rewriting or optimization) or evaluate them (e.g., triple stores). In such projects, SPARQL frequently occurs as part of testing or benchmark components. Projects may also include query fragments as templates or for query generation.
- **C3**. **Reasoning Tools**. Applications that include SPARQL queries, but not for the purpose of querying data stores, but for its reasoning capabilities. Practical examples include tasks such as rule-based data validation.

We are not interested in these kinds of applications in the context of our second research questions. Instead, we aim to analyze usage of SPARQL in concrete, data centric applications, defined as follows:

- **C4**. **Applications**. Applications using SPARQL to query for data. Query usage in such applications can be further categorized as:
  a. **Meta**. Queries predominantly explore the structure of a data source, i.e., select for properties and concepts. Example: Quality checker for SKOS vocabularies.
  b. **Concrete**. Queries predominantly select concrete instances of concepts. Example: Management system for the Oslo public library.

Since we aim to investigate applications working on concrete instance data, we are interested in projects belonging to category C4.b.

Such applications might exhibit, and thus be identified by, a number of features. In order to find such concrete applications, we extend our general approach with two further steps, as is outlined in Figure 3. We first apply more restrictive filters to $\mathcal{D}_{queries}$ (see also Additional Filters below) and then rank remaining projects based on the log-scaled and normalized average of the following three features, before manually reviewing the top ranking projects.

LOV Concrete queries might rely on ontologies or other vocabularies. 660 popular linked open data vocabularies are listed in the LOV [32, 67] data set. In particular, it provides URIs of vocabularies which, for example, can be part of prefix definitions or occur in SPARQL queries. We search all Java files and count occurrences of URIs that occur in the data set. We limit our analysis to Java source files, since other files may use these URIs to describe arbitrary artifacts (e.g., in XML files) unrelated to query usage.

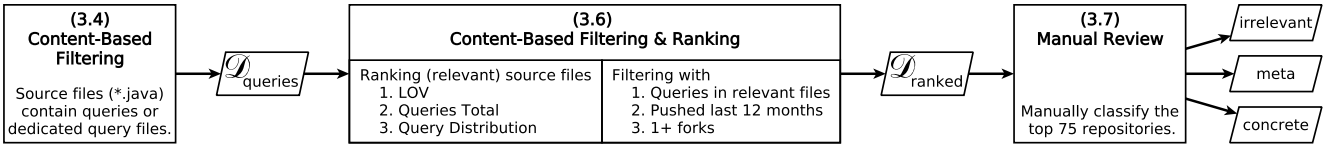

Figure 3. Overview of our extended approach (related to RQ2) as an extension to Figure 2.

Number of Queries Secondly, we consider the total number of SPARQL queries in relevant source files. We define source files as relevant, if they match the following criteria. Firstly, the file must be a Java source file In addition, the name of the file must not include the term `test` (case-insensitive) and the path of the file within the repository must not include the terms `test` or `testing` (case-insensitive). Finally, the path must also not include the terms `example` or `examples` (case-insensitive). With this definition of *relevant* source files, we aim to exclude projects which include SPARQL (exclusively) in their tests or usage examples. By strong convention, Java tests use the word 'test' in their class (and file) name and are located in a folder such as `src/test`. Even in instances where these conventions are not followed (e.g., when the project does not use a common test framework), paths to such test cases are still likely to include the terms `test` or `testing`. Similarly, we aim to exclude usage examples. As strong a convention as for tests does not exists in this case, so we only exclude directories that explicitly include the term `example`. For all relevant source files, we search for SPARQL headers using the same approach as before (Section 3.4) and we again include dedicated query source files.

Query Distribution Our third feature refers to the percentage of relevant files that contain SPARQL queries, using the same criteria for relevance as for the Number of Queries. This is calculated as the count of relevant files that include at least one SPARQL query (i.e., matched by the second feature) over the count of all relevant files.

Additional Filters On top of the ranking calculated based on the three features above, we apply a number of additional filters to the data set: The Number of Queries (and as a consequence Query Distribution) must be non-zero. This eliminates projects without queries in *relevant* source files. We also apply additional popularity filters to reduce the overall number of repositories for manual review. In addition to two or more stars, projects must also have a push within the last 12 months and at least one fork.

3.7 Manual Review

The ranking and filters we have defined in the previous step produce a greatly reduced number of ranked candidates. We manually review the top 75 highest ranked repositories and label them as either *irrelevant*, *meta* or *concrete* in the following manner. A reviewer first reads the description (which includes the primary `README.md` file and the project description) of the repository. If available (i.e., linked in any part of the project description as defined above), external project websites are considered as well. A short comment about the nature of the repository is recorded. In a second step, the reviewer investigates the usage of SPARQL within the project by considering (examples of) detected SPARQL queries that exist in the project. In this second step, *irrelevant* projects can be identified via their use of SPARQL in a reasoning context or solely as part of examples or test cases that were missed by our filters. We label all projects pertaining to categories C1 through C3 as *irrelevant*. Examples include graph database implementations, various optimization approaches such as SPARQL to SQL rewriting and validation rules expressed in SPARQL, among others. We only report the number of repositories labeled as such, but do not consider them any further.

Projects that were not immediately labeled as *irrelevant* are examined more closely, in order to differentiate between meta query usage versus concrete queries (categories C4.a and C4.b) and projects are then labeled accordingly. For applications labeled as *meta*, some overlap with *irrelevant* exists. For example, various frameworks use SPARQL to obtain meta-level information about ontologies or resources they manage. In such cases, we prefer the *meta* label. The *concrete* label is used for applications matching exactly category C4.b. Examples for this category are discussed in more detail with the results of our second research question.

4 Results

The results presented in this paper are based on an execution of our query pipeline in June, 2019. The resulting data sets are available in a git repository[3].

4.1 First Research Question

For our first research question, we compare the overall popularity of graph-based query languages. Figure 4 shows the total number of distinct Java repositories returned for the dependencies related to each of the four graph-related technologies as a Venn diagram. These repositories are not filtered by popularity, but exclude any repositories that do not employ Java (the $\mathcal{D}_{total}$ data set). We find that the most related repositories exist for Cypher (9882), followed by SPARQL (6919). In the Venn diagram, overlap means that repositories

[3] https://www.github.com/softlang/graphqls

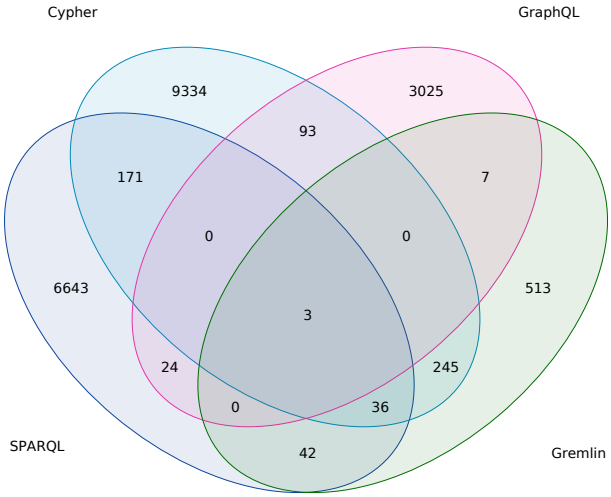

Figure 4. Venn diagram of repositories per language.

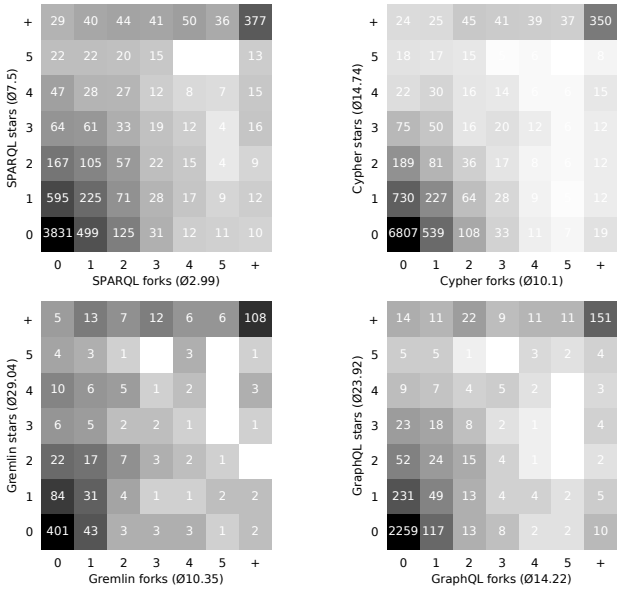

Figure 5. Popularity metrics: Number of forks and stars ranging from 0 to 5, or more than 5 (+) as well as the average number forks and stars (Ø). Colors indicate percentage of repositories on a *logarithmic* scale. Correlation coefficients are 0.86, 0.95, 0.88 and 0.84 respectively.

are in the result sets of both overlapping languages (note, that the diagram is not rendered to scale). Overlap primarily exists between Gremlin and SPARQL, as well as Gremlin and Cypher. In fact, 40% of Gremlin repositories are related to at least one additional language, with the largest overlap existing between Cypher and Gremlin. Cypher also overlaps with GraphQL. This is not unexpected, as for example Neo4j supports both traversal via Gremlin and the GraphQL language, in addition to Cypher. While some minimal overlap between our initial dependencies exists (a total of 5 dependencies are shared among two or more languages), this accounts for only a handful of overlapping repositories, to a maximum of five of the overlap between Gremlin and Cypher. The most frequently occurring dependencies (i.e., most projects in our data set depend on them) are `org.apache.jena` followed by `org.eclipse.rdf4j` for SPARQL, `org.neo4j` followed by `org.opencypher` for Cypher, `org.apache.tinkerpop` for Gremlin and `com.graphql-java` for GraphQL.

Popularity In order to filter less relevant repositories from our data set, we consider the frequently used popularity metrics of stars. In Figure 5, both the number of stars and forks for all four languages are visualized. Most repositories have 0 forks and 0 stars. The averages indicate a small number of extremely popular repositories, while the majority has few or no stars. Accordingly, the median number of stars and forks (excluding 0) is 2 for all languages except Gremlin, where it is 4 and 3, respectively. The correlation coefficient for both metrics is strong (as expected, e.g. [11]) and ranges between 0.84 for GraphQL and 0.95 for Cypher. We choose a minimum of 2 stars for our popularity restricted data set in RQ1, while additionally requiring 1 fork for RQ2.

Number of Projects Figure 6 (first row) depicts the development of repositories in $\mathcal{D}_{total}$ over time, where the x-axis is time and the y-axis describes the count of projects that existed at this time. On the right hand side of the first row, the same is depicted for popular projects with 2 or more stars. Developing almost identically before, over the last three years Cypher related projects overtake SPARQL. This closely correlates with the establishment of the openCypher group in late 2015, possibly explaining this change in number of projects. There is less of a difference in project numbers for popular projects, with SPARQL taking a slight lead. For both Gremlin and GraphQL, significantly fewer projects exist, both popular and in general. The number of non-popular GraphQL projects, however, grows rapidly since its publication. We must note, that these numbers might be biased by the fact that we only compare Java applications. Since GraphQL is primarily a REST replacement, projects in other languages, such as JavaScript, might exist.

Baseline We limit the scope of our baseline comparison to repositories using the JDBC driver of the most popular SQL database available through Maven. We define popularity according to `db-engines.com` [64]. The highest ranking system that matches our criteria is *MySQL*, the second highest ranking database overall. We select repositories that use the official JDBC driver of MySQL (`mysql-connector-java`). In

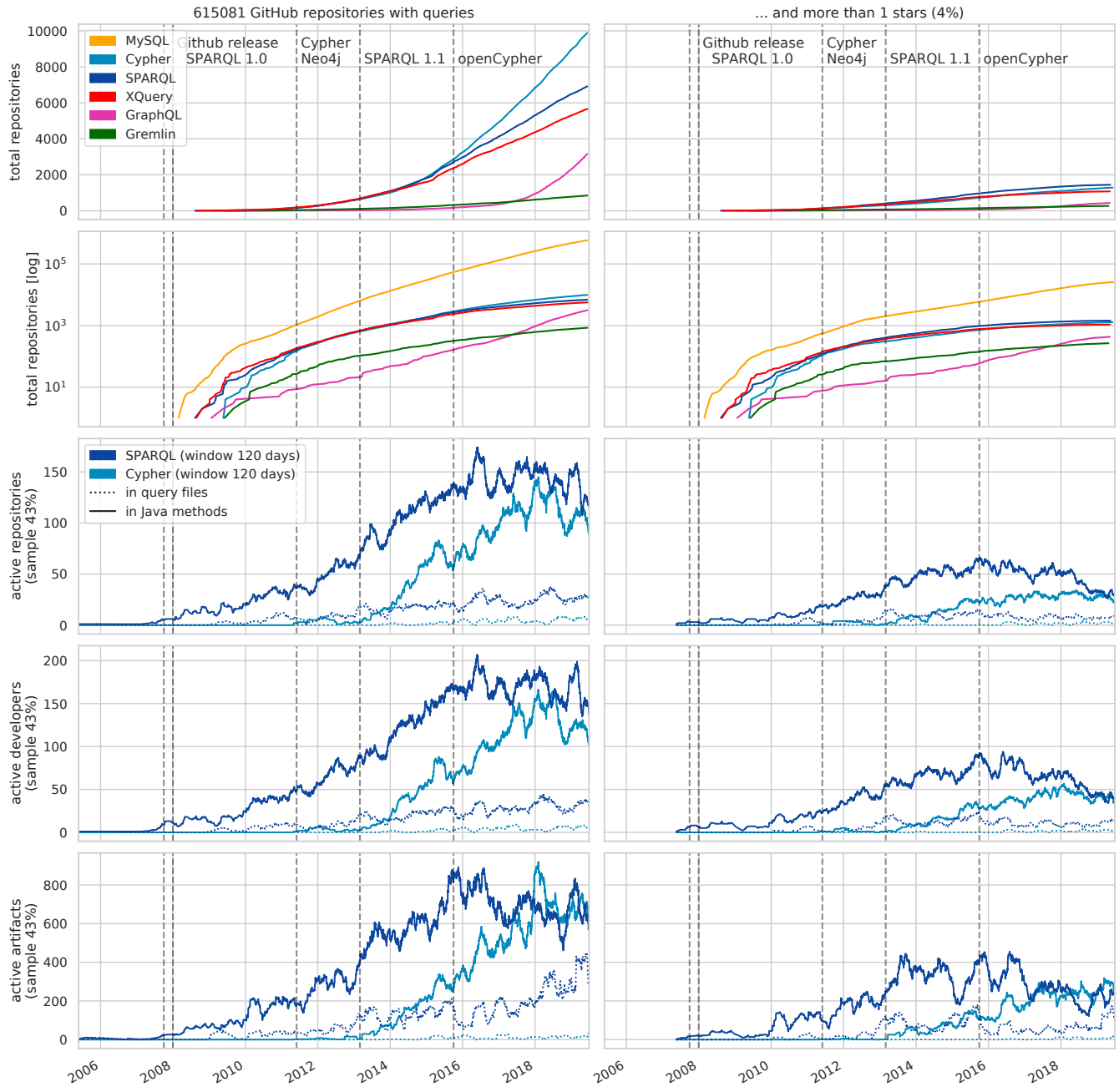

Figure 6. The total amount of repositories and the activity concerned with query languages: The left column shows all repositories, the right filters on those with more than one star. The first and second rows show the total amount of repositories for all query languages. Following rows show the repository activity computed on a 43% sample of all repositories. It shows the number of distinct repositories, developers and artifacts involved in a query activity in the last 120 days. We add important GitHub, SPARQL and Cypher related dates: Github release (Oct 2007); W3C Recommendation of SPARQL 1.0 (Jan 2008); Introduction Cypher to Neo4j (Jun 2011); W3C Recommendation of SPARQL 1.1 (Mar 2013); openCypher founded (Oct 2015).

addition to SQL, we also compare with the XQuery language, where we use the same approach as for the graph-related languages. Figure 6 (second row) shows the development of project numbers over time for MySQL, XQuery and the graph related query languages. Not unexpectedly, MySQL is significantly more popular than any other language we compare, while popularity of XQuery is comparable to SPARQL and Cypher, being overtaken slightly over the last few years.

Query Activity Existing repository can stop being maintained or queries can become outdated without actually being deleted. Therefore, we complement the information on the total amount of repositories existing at a certain time, as shown previously in Figure 6 (row one), by a measure on query activity based on a repository's commits.

The activity in respect to queries is depicted in the last three rows of Figure 6, showing data gathered on a sample (43%) of the overall repositories featuring the two most popular query languages SPARQL and Cypher[4]. We compute the temporal evolution of the number of distinct repositories, developers and artifacts involved in a query activity in the last 120 days. An activity is a commit changing a query related artifact. The analysis depicted in Figure 6 further distinguish between activity induced by queries in dedicated query files (dotted lines) and such done in Java methods (straight lines). Further, we plot the evolution for all repositories (left column) and for those with 2 or more stars (right column). We make the following observations:

- SPARQL queries have been used more actively compared to Cypher queries until 2018, with respect to the involved repositories, developers and artifacts.
- Cypher activity increased faster compared to SPARQL activity. Cypher has been introduced to Neo4j three years later than the initial recommendation of SPARQL 1.0 in 2008; however, the activity is almost on par in 2019.
- Java embedding is more prominent than using dedicated query files for SPARQL and Cypher. We observe this in the numbers of distinct repositories and developers working actively with files or methods. We do not draw this conclusion from the number of distinct artifacts active, as the granularity of methods and query files may vary. Dedicated query files are much more uncommon for Cypher.
- We notice a stagnation in the last year respective active repositories and developers for both query technologies. The activity reaches its peak at 2018 and does not further increase. For repositories rated better than 1 star the SPARQL activity stops improving after 2016.

Repository Characteristics Differences in the repository characteristics for SPARQL and Cypher are examined by a linear regression applied on the 43% sample of repositories containing one of both query technologies. We use stars, developers actively working with queries, query files and the time since repository creation as independent variables and regress on the dependent variable *is_sparql* (i.e., whether a repository contains SPARQL queries opposed to not containing SPARQL but Cypher). We interpret the partial regression coefficients on significance and sign as indication for a higher or lower characteristic for SPARQL in the population at hand.

Table 3. Linear regression on whether a repository contains SPARQL queries. The regression is applied on the 43% sample of repositories containing SPARQL or Cypher. The table shows the partial regression coefficients and the significance of the coefficients for three models.

dep. var: is_sparql	Model 0	Model 1	Model 2
scale(np.log(active_developers + 0.5))		0.134***	0.117***
scale(np.log(active_files + 0.5))			0.06***
scale(np.log(created_days_ago))	0.038***	0.04***	0.04***
scale(np.log(stargazers_count + 0.5))	0.043***	0.019**	0.017**
R-squ.	0.017	0.089	0.102

(*** p < 0.001, ** p < 0.01, * p < 0.05)

Table 3 depicts the partial regression coefficients in combination with the significance levels. We log-transform and scale the independent variables to assure normality. We successively construct three models adding the factors on the repository meta-data and the query activity[5]. We report on r-square for model fit. We draw the following conclusions:

- SPARQL repositories tend to exist longer than Cypher repositories, as the regression coefficients are positive and significant in all three models for *created_days_ago*. This reflects the SPARQL and Cypher release dates.
- The coefficients for active query files are positive and significant, indicating that for SPARQL repositories contain more queries in files compared to Cypher.
- Accordingly, the partial regression coefficients for the active developers working with queries are positive and significant. Hence, we conclude that SPARQL repositories tend to involve more developers working on queries than Cypher repositories do. However, Cypher manages to keep on par in the history of query activity, as it exceeds SPARQL in the number of repositories.
- We can not make highly significant statements on the ratings of repositories in this regression; but Cypher repositories tend to share a lower rating than those involving SPARQL.

4.2 Second Research Question

After applying our additional popularity filters, we obtain a reduced number of 475 repositories. When applying our ranking and filtering (concerning queries in relevant artifacts), we obtain the data set $\mathcal{D}_{ranked}$, consisting of 150 repositories. We review the top ranking 75 (50%) manually. This review

[4]In total, we processed 7274 repositories excluding 2.7% due to a timeout for the processing of each repository. Our data set contains 1373.000 analyzed commits; 22.000 contain queries; we excluded commits with more than 32 changed artifacts concerned with queries (1.21% of 22.000 commits). We manually adjusted this threshold to avoid unnatural steps induced by single commits.

[5]All mutual correlations of the independent variables are under 0.3 and the variance inflation factors are all under 2, suggesting no problems with multicollinearity.

Table 4. Repositories labeled *concrete* with a short description summarizing the application. Results in ranking order (#1 = highest relative rank). KnowledgeCaptureAndDiscovery was shortened to KCAD in this table.

#	Repository	Description
1	sirmaenterprise/conservation-space	Content management for conservators.
2	GRIDAPPSD/Powergrid-Models	Model conversion in context of grid simulation models.
3	vivo-project/Vitro	Semantic web application framework (that uses SPARQL).
4	linkedpipes/etl	RDF-based ETL tool.
5	WDAqua/SummaServer	Service for summarizing (specific) RDF graphs.
6	RENCI-NRIG/orca5	Meta-cloud managing network description model.
7	oeg-upm/map4rdf	Visualization and interaction for linked, geospatial data.
8	digibib/ls.ext	Oslo public library management system.
9	KCAD/wings	Computational experiment workflow design system.
10	KCAD/DataNarratives	Text-based narrative generation for workflow results.
11	rmap-project/rmap	Management system of complex scholarly contribution relations.
12	EBISPOT/goci	Visualization component of GWAS catalog curation tool.

results in 12 *concrete*, 21 *meta* and 42 *irrelevant* repositories. Projects labeled as *irrelevant* range from frameworks or framework extensions, graph databases, specific API wrappers, to coursework submissions and personal blogs that include SPARQL tutorials. In these cases, queries occur as examples, test cases or benchmarks, usually in a non-standard way that evaded our filters.

A good examples of the *meta* label is cmader/qSKOS, a tool for finding quality issues in SKOS vocabularies. The project uses plenty of SPARQL queries as well as the SKOS vocabulary. However, queries in this project operate on the meta level and do not select concrete instance data. Other meta uses of SPARQL include language generation or question answering applications.

In order to answer our second research question, we now consider the repositories labeled as *concrete*, which are listed in their relative ranking order in Table 4, together with a short description of the application that we recorded during the labeling process. We will henceforth use the placement in Table 4 to refer to these projects in other figures.

Domains The 12 applications can be grouped into specific and general application domains, as is shown in Table 5. We find that most applications are management related, with 3 document management systems and 3 process management systems. Of the remaining 6 applications, 4 are related to data visualization as either images (geospatial data) or text. The remaining 2 are a tool for model conversion and a web application framework that do not fit into any of the three general categories.

SPARQL Figure 7 shows the distribution of SELECT (returns result sequences), ASK (returns Boolean value), DESCRIBE (returns specific graphs) and CONSTRUCT (returns graphs) queries over the 12 SPARQL applications. Only four repositories use more than 50 queries, with the average being 73. The most frequent query kind is SELECT, with half of the applications using exclusively SELECT queries. DESCRIBE queries are only used in two projects, and no project uses DESCRIBE without also using CONSTRUCT queries. Half of all projects use ASK queries in addition to SELECT (and other) query kinds.

Table 5. Application domains of the top 12 applications labeled as *concrete*. The number (#) refers to Table 4.

#	Domain	Specific Domain
1	Document Management	Conservation
8	Document Management	Library
11	Document Management	Scholarly/Authorship
4	Process Management	ETL Process
6	Process Management	Cloud Management
9	Process Management	Workflow
5	Data Visualization	Summarization
7	Data Visualization	Geospatial Data
10	Data Visualization	Workflow Narrative
12	Data Visualization	GWAS
2	Simulation	Model Conversion
3	Web Application	Web App Framework

Table 6. Ontology usage in *concrete* labeled repositories, with PS referring to the existence of project specific ontologies (+) or lack thereof (−), DS the count of domain specific ontologies and G the count of generic vocabularies.

	1	2	3	4	5	6	7	8	9	10	11	12
PS	+	-	+	+	-	+	+	+	+	-	+	+
DS	9	2	2	1	9	3	6	1	3	5	2	5
G	4	4	6	2	4	5	3	4	1	2	1	5

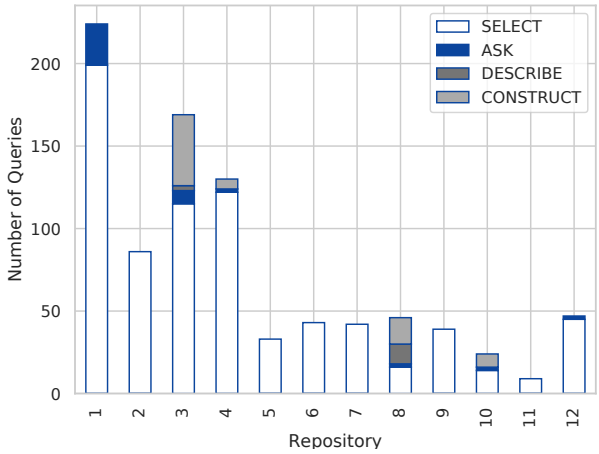

Figure 7. Number of queries divided by query kind for the 12 applications labeled as *concrete*.

Ontologies Table 6 shows usage of ontologies across all *concrete* projects. Interestingly, most projects employ three kinds of ontologies: (1) Project specific ontologies, that is ontologies that were specifically designed for the project. (2) Domain specific or specialized ontologies, that relate to the application domain of the project (but were not created specifically for it), and (3) general vocabularies, such as OWL, RDF or RDFS. For project specific ontologies we only record whether such ontologies exist, not the quantity. For domain specific and generic vocabularies, we record a lower bound of quantity, that was automatically extracted by search for URLs occurring in the LOV data set as well as searching source code for SPARQL prefix definitions. All applications use domain specific ontologies and generic vocabularies. Only three applications do not employ any project specific ontologies, two of which are in the visualization domain. The third (#2) is the model conversion tool, which uses a very specific domain related ontology.

4.3 Threats to Validity

We only include repositories that directly depend on libraries that can be found on `mvnrepository` via the name of the language. The limitations are twofold: For one, not all existing libraries related to the technologies might be found this way. In particular, we might not find proprietary frameworks via Maven. We argue, however, that such frameworks are unlikely to be used frequently in open-source projects. Secondly, we do not select repositories that only transitively depend on these dependencies.

Composition of queries as strings is not standardized between frameworks, or even within a single framework. Instead, projects rely on various string composition techniques available in Java. Our query extraction aims to generalize to common scenarios, but there might be exceptions that are missed by our filters.

Finally, our results regarding application domains, query and ontology usage in SPARQL projects include primarily smaller, experimental or research applications. All projects considered in our study are open-source applications available on GitHub. Our results, therefore, can not necessarily be transferred to industry usage or other scenarios beyond the scope of such open-source projects.

5 Related Work

Graph Query Languages and Databases Graph query languages and databases have been studied regarding various aspects. In a study on their theoretical properties [4], the authors compare graph data models (property and edge-labeled graphs) as well as querying approaches (graph patterns and navigational expressions), referencing examples in SPARQL, Cypher and Gremlin. Similar surveys were performed in the past [6, 33, 72]. Cypher and Gremlin have also been compared in terms of their performance when accessing Neo4j [34]. Graph databases and database models have been compared in various studies [2, 36]. In [69], the authors compare the relational model (MySQL) with the graph model (Neo4j), in terms of performance and subjective measures, such as ease of use. A similar comparison is presented in [47]. Our work presents an empirical study on the usage of graph query languages in open-source projects.

Cross-Sectional Studies The amount of publicly available software projects on GitHub or SourceForge yet represents an accurate picture of the current and past of open-source development. Several empirical studies founded on such data analyze various aspects of the involved technology. In the following, we focus on cross-sectional observations in empirical studies, i.e., on observations done at a certain time. In [17], aspects such as co-occurrence of relational database access technologies are examined on 2,457 Java projects hosted on GitHub. The repositories are selected by searching for commits with references to jdbc, jpa or hbm. We identify projects via dependencies in their build files. While we analyze ontology usage, the authors of [39] examined GraphQL Schema usage in 20.000 GitHub projects. GraphQL Schemas where also studied in [71], while comparing usage in small and large GitHub projects, as well as commercial projects. The research reported on in [58] faces a related problem to our detection of embedded queries in Java code, namely the reconstruction of link URLs in Andorid apps. A total of 5.000 repositories are analyzed for the usage of Mocking frameworks in [48]; test driven development in 256.000 repositories is subject to analysis in [12]; the usage of asserts in the 100 most popular C or C++ repositories in revises in [13]. All such analysis is working on GitHub repositories. In [46], details like the costs or practices in local database usage are examined. The corpus for analysis consists of the top

1.000 ranked Google Play apps. In [43, 44], the authors analyze usage of the P3P privacy policy language in respect to vocabulary, correctness, common policies and policy clones. Here, the corpus was constructed from websites listed in the Google Directory and Open Directory Project, respectively. The authors of [31] report on the characteristics of 2.080 randomly chosen Java applications from SourceForge. Findings are, for instance, that only a few Java methods are overridden. The popularity, interoperability and the impact of different languages are analyzed based on a sample of 100.000 GitHub projects in [10]. Opposed to that, [50] compares languages on the Rosetta Code corpus. The uniqueness of source code is studied in [28] on 6.000 SourceForge software projects. The empirical study in [66] focuses on social factors affecting the evaluation of GitHub contributions. Here the analyzed corpus consists of accepted and rejected pull requests in GitHub projects. In [68], the gender and tenure diversity on GitHub teams is discussed. A wide range of empirical studies is concerned with the usage of API. Such studies analyze the usage [45, 61, 63, 73, 75], API migration [62] and API patterns [76].

Longitudinal Studies Due to the availability of the full repository history in open-source projects, longitudinal studies are frequently applied, so that trends in the evolution of technologies can be recovered. In [54], the evolution of exceptions (e.g., the amount of custom exceptions) is examined in four open-source Java systems. In [74], the evolution of testing libraries is subject to analysis showing the migrations in between such libraries. In [65], the evolution of unit tests in 70.000 revisions of 10 different Python projects is examined. In [41], model driven engineering technologies in the context of EMF are analyzed, including the evolution of the number of commits and contributing developers. We also capture such data respective query usage. In [55], the evolution and correlation of code-smells on 395 releases of 30 software systems is discussed. The authors of [1] examine the co-evolution of documentation related artifact types and the repository popularity. In particular, the authors track ratings over time; we rely on the most recent rating of a repository. In [35], evolutionary coupling is detected using association rules. The authors of [14] examine co-commit patters in GitHub repositories. The longitudinal part of our study analyzes 7274 repositories, including the history in terms of 1373.000 commits, thereby covering the development activity of SPARQL and Cypher in the past years.

Mining Software Repositories Analysis of open-source repositories is also subject to more general research in the context of software linguistics [22]. In [9, 37, 38], the promises and perils of mining GitHub's open-source data for empirical analysis is discussed. The effects of branching in the repository history on empirical studies is discussed in [42]. We treat branching by a dedicated method. In [15], a meta-study summarizes empirical studies applied to GitHub with respect the methods, data set and limitation. One finding is that only a few studies are longitudinal; our detailed analysis of the commits history falls into such classification. The dangers of unappropriated partitioned commits are discussed in [19, 40]. We apply a threshold for exuding commits as outliers. Identity management of developers is revised in [8]. We go for a conservative solution assuming each email address to belong to one developer.

6 Conclusion and Future Work

In this study we investigated the popularity of graph-related query languages in open-source projects on GitHub. We find evidence for the real-world usage of graph query technologies. Of the languages we compare, SPARQL and Cypher are significantly more frequent than other graph-related approaches. Both are slightly more popular than XQuery, but all approaches we compare are significantly less popular than SQL. In a direct comparison between SPARQL and Cypher, we observe higher activity in SPARQL related repositories in terms of query related changes to repositories and artifacts, as well as active developers. However, activity in Cypher grew faster, almost reaching the same level of activity as SPARQL in 2019, even though Cypher was introduced three years after SPARQL. For both languages, the activity regarding repositories and developers stagnates after a peak in 2018. For popular SPARQL repositories with 2 or more stars, this occurs as early as 2016.

We also analyzed the usage of SPARQL in concrete applications. Only a small fraction of popular SPARQL projects are applications that employ SPARQL to query for concrete data, instead of frameworks or projects focusing on graph structure related queries. Applications that do exist are predominately related to document or process management tasks, or visualize data as images or text. SELECT queries are the most frequently used kind of SPARQL query in these applications. Projects do extensively employ ontologies, simultaneously creating projects specific ones and reusing domain specific and general vocabularies.

For future work, we plan to perform an in-depth analysis of developer interaction with query related artifacts on the project level, including the specialization of developers and the relationship between query artifacts and bug fixes. We are also interested in comparing the usage of embedded queries with approaches using fluent APIs. Furthermore, we are interested in application of our commit history evaluation to additional languages such as SQL, and the inclusion of additional query languages to our overall approach.

Acknowledgments

The authors gratefully acknowledge the financial support of project LISeQ (LA 2672/1-1) by the German Research Foundation (DFG).

References

[1] Karan Aggarwal, Abram Hindle, and Eleni Stroulia. 2014. Co-evolution of project documentation and popularity within github, See [18], 360–363. https://doi.org/10.1145/2597073.2597120

[2] Renzo Angles. 2012. A Comparison of Current Graph Database Models. In *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 171–177. https://doi.org/10.1109/ICDEW.2012.31

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages, See [16], 1421–1432. https://doi.org/10.1145/3183713.3190654

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. https://doi.org/10.1145/3104031

[5] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. 2004. An Automatic Approach to identify Class Evolution Discontinuities. In *7th International Workshop on Principles of Software Evolution (IWPSE 2004), 6-7 September 2004, Kyoto, Japan*. IEEE Computer Society, 31–40. https://doi.org/10.1109/IWPSE.2004.1334766

[6] Pablo Barceló Baeza. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, Richard Hull and Wenfei Fan (Eds.). ACM, 175–188. https://doi.org/10.1145/2463664.2465216

[7] Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). 2015. *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7174815

[8] Christian Bird, Alex Gourley, Premkumar T. Devanbu, Michael Gertz, and Anand Swaminathan. 2006. Mining email social networks, See [20], 137–143. https://doi.org/10.1145/1137983.1138016

[9] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. Germán, and Premkumar T. Devanbu. 2009. The promises and perils of mining git. In *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings*, Michael W. Godfrey and Jim Whitehead (Eds.). IEEE Computer Society, 1–10. https://doi.org/10.1109/MSR.2009.5069475

[10] Tegawendé F. Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. 2013. Popularity, Interoperability, and Impact of Programming Languages in 100, 000 Open Source Projects. In *37th Annual IEEE Computer Software and Applications Conference, COMPSAC 2013, Kyoto, Japan, July 22-26, 2013*. IEEE Computer Society, 303–312. https://doi.org/10.1109/COMPSAC.2013.55

[11] Hudson Borges, André C. Hora, and Marco Tulio Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 334–344. https://doi.org/10.1109/ICSME.2016.31

[12] Neil C. Borle, Meysam Feghhi, Eleni Stroulia, Russell Greiner, and Abram Hindle. 2018. Analyzing the effects of test driven development in GitHub. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1062. https://doi.org/10.1145/3180155.3182535

[13] Casey Casalnuovo, Premkumar T. Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. 2015. Assert Use in GitHub Projects, See [7], 755–766. https://doi.org/10.1109/ICSE.2015.88

[14] Eldan Cohen and Mariano P. Consens. 2018. Large-scale analysis of the co-commit patterns of the active developers in github's top repositories. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 426–436. https://doi.org/10.1145/3196398.3196436

[15] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2016. Findings from GitHub: methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, Miryung Kim, Romain Robbes, and Christian Bird (Eds.). ACM, 137–141. https://doi.org/10.1145/2901739.2901776

[16] Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). 2018. *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM. https://doi.org/10.1145/3183713

[17] Alexandre Decan, Mathieu Goeminne, and Tom Mens. 2015. On the Interaction of Relational Database Access Technologies in Open Source Java Projects. In *Post-proceedings of the 8th Seminar on Advanced Techniques and Tools for Software Evolution, Mons, Belgium, July 6-8, 2015. (CEUR Workshop Proceedings)*, Anya Helene Bagge, Tom Mens, and Haidar Osman (Eds.), Vol. 1820. CEUR-WS.org, 26–35. http://ceur-ws.org/Vol-1820/paper-03.pdf

[18] Premkumar T. Devanbu, Sung Kim, and Martin Pinzger (Eds.). 2014. *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. ACM. http://dl.acm.org/citation.cfm?id=2597073

[19] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik (Eds.). IEEE Computer Society, 341–350. https://doi.org/10.1109/SANER.2015.7081844

[20] Stephan Diehl, Harald C. Gall, and Ahmed E. Hassan (Eds.). 2006. *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*. ACM. https://doi.org/10.1145/1137983

[21] Inc. Facebook. 2019. *GraphQL Specification*. Retrieved June 27, 2019 from https://graphql.github.io/graphql-spec/

[22] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. 2010. Empirical Language Analysis in Software Linguistics. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers (Lecture Notes in Computer Science)*, Brian A. Malloy, Steffen Staab, and Mark van den Brand (Eds.), Vol. 6563. Springer, 316–326. https://doi.org/10.1007/978-3-642-19440-5_21

[23] The Apache Software Foundation. 2019. *Apache Jena ARQ Documentation*. Retrieved June 27, 2019 from http://jena.apache.org/documentation/query/index.html

[24] The Apache Software Foundation. 2019. *Apache TinkerPop*. Retrieved June 27, 2019 from https://tinkerpop.apache.org

[25] The Apache Software Foundation. 2019. *Gremlin Overview*. Retrieved June 27, 2019 from https://tinkerpop.apache.org/gremlin.html

[26] The Apache Software Foundation. 2019. *The Central Repository*. Retrieved June 27, 2019 from https://repo.maven.apache.org/maven2/

[27] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs, See [16], 1433–1445. https://doi.org/10.1145/3183713.3190657

[28] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and André van der Hoek (Eds.). ACM, 147–156. https://doi.org/10.1145/1882291.1882315

[29] GitHub. 2019. *Linguist Repository.* Retrieved June 27, 2019 from https://github.com/github/linguist/blob/master/lib/linguist/languages.yml

[30] GraphQL Foundation. 2019. *GraphQL Foundation.* Retrieved June 27, 2019 from https://gql.foundation/

[31] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi-Reghizzi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. 2010. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM 2010, 16-17 September 2010, Bolzano/Bozen, Italy*, Giancarlo Succi, Maurizio Morisio, and Nachiappan Nagappan (Eds.). ACM, Article 11, 10 pages. https://doi.org/10.1145/1852786.1852801

[32] Ontology Engineering Group. 2019. *Linked Open Vocabularies.* Retrieved June 27, 2019 from https://lov.linkeddata.es

[33] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. 2004. A Comparison of RDF Query Languages. In *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings (Lecture Notes in Computer Science)*, Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen (Eds.), Vol. 3298. Springer, 502–517. https://doi.org/10.1007/978-3-540-30475-3_35

[34] Florian Holzschuher and René Peinl. 2013. Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. In *Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*, Giovanna Guerrini (Ed.). ACM, 195–204. https://doi.org/10.1145/2457317.2457351

[35] Md. Anaytul Islam, Md. Moksedul Islam, Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2018. [Research Paper] Detecting Evolutionary Coupling Using Transitive Association Rules. In *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018.* IEEE Computer Society, 113–122. https://doi.org/10.1109/SCAM.2018.00020

[36] Salim Jouili and Valentin Vansteenberghe. 2013. An Empirical Comparison of Graph Databases. In *International Conference on Social Computing, SocialCom 2013, SocialCom/PASSAT/BigData/EconCom/BioMedCom 2013, Washington, DC, USA, 8-14 September, 2013.* IEEE Computer Society, 708–715. https://doi.org/10.1109/SocialCom.2013.106

[37] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. 2014. The promises and perils of mining GitHub, See [18], 92–101. https://doi.org/10.1145/2597073.2597074

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

[39] Yun Wan Kim, Mariano P. Consens, and Olaf Hartig. 2019. An Empirical Analysis of GraphQL API Schemas in Open Code Repositories and Package Registries. In *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay, June 3-7, 2019. (CEUR Workshop Proceedings)*, Aidan Hogan and Tova Milo (Eds.), Vol. 2369. CEUR-WS.org. http://ceur-ws.org/Vol-2369/short04.pdf

[40] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! are you committing tangled changes?. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, Chanchal K. Roy, Andrew Begel, and Leon Moonen (Eds.). ACM, 262–265. https://doi.org/10.1145/2597008.2597798

[41] Dimitrios S. Kolovos, Nicholas Drivalos Matragkas, Ioannis Korkontzelos, Sophia Ananiadou, and Richard F. Paige. 2015. Assessing the Use of Eclipse MDE Technologies in Open-Source Software Projects. In *Proceedings of the International Workshop on Open Source Software for Model Driven Engineering co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 29, 2015. (CEUR Workshop Proceedings)*, Francis Bordeleau, Jean-Michel Bruel, Juergen Dingel, Sebastien Gerard, and Sebastian Voss (Eds.), Vol. 1541. CEUR-WS.org, 20–29. http://ceur-ws.org/Vol-1541/OSS4MDE_2015_paper_2.pdf

[42] Vladimir Kovalenko, Fabio Palomba, and Alberto Bacchelli. 2018. Mining file histories: should we consider branches?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 202–213. https://doi.org/10.1145/3238147.3238169

[43] Ralf Lämmel and Ekaterina Pek. 2010. Vivisection of a Non-Executable, Domain-Specific Language - Understanding (the Usage of) the P3P Language. In *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010.* IEEE Computer Society, 104–113. https://doi.org/10.1109/ICPC.2010.45

[44] Ralf Lämmel and Ekaterina Pek. 2013. Understanding privacy policies - A study in empirical analysis of language usage. *Empirical Software Engineering* 18, 2 (2013), 310–374. https://doi.org/10.1007/s10664-012-9204-1

[45] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. 2011. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung (Eds.). ACM, 1317–1324. https://doi.org/10.1145/1982185.1982471

[46] Yingjun Lyu, Jiaping Gui, Mian Wan, and William G. J. Halfond. 2017. An Empirical Study of Local Database Usage in Android Applications. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017.* IEEE Computer Society, 444–455. https://doi.org/10.1109/ICSME.2017.75

[47] Justin J Miller. 2013. Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA.* 24.

[48] Shaikh Mostafa and Xiaoyin Wang. 2014. An Empirical Study on the Usage of Mocking Frameworks in Software Testing. In *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014.* IEEE, 127–132. https://doi.org/10.1109/QSIC.2014.19

[49] MvnRepository. 2019. *MvnRepository Search Engine.* Retrieved June 27, 2019 from https://mvnrepository.com

[50] Sebastian Nanz and Carlo A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code, See [7], 778–788. https://doi.org/10.1109/ICSE.2015.90

[51] Neo4j, Inc. 2019. *Neo4j Product Website.* Retrieved June 27, 2019 from https://neo4j.com/

[52] openCypher Project. 2018. *OpenCypher Version 9 Standard.* Retrieved June 27, 2019 from https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf

[53] openCypher Project. 2018. *Project Website.* Retrieved June 27, 2019 from http://www.opencypher.org/

[54] Haidar Osman, Andrei Chis, Jakob Schaerer, Mohammad Ghafari, and Oscar Nierstrasz. 2017. On the evolution of exception usage in Java projects, See [56], 422–426. https://doi.org/10.1109/SANER.2017.7884646

[55] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information & Software Technology* 99 (2018), 1–10.

[56] Martin Pinzger, Gabriele Bavota, and Andrian Marcus (Eds.). 2017. *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017.* IEEE Computer Society. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7879528

[57] Eric Prud'hommeaux and Andy Seaborne. 2013. *SPARQL Query Language for RDF.* Retrieved June 27, 2019 from https://www.w3.org/TR/rdf-sparql-query/

[58] Marianna Rapoport, Philippe Suter, Erik Wittern, Ondrej Lhoták, and Julian Dolby. 2017. Who you gonna call?: analyzing web requests in Android applications. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 80–90. https://doi.org/10.1109/MSR.2017.11

[59] Jonathan Robie, Michael Dyck, and Josh Spiegel. 2017. *XQuery 3.1: An XML Query Language* . Retrieved June 27, 2019 from https://www.w3.org/TR/xquery-31/

[60] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language. *CoRR* abs/1508.03843 (2015). arXiv:1508.03843 http://arxiv.org/abs/1508.03843

[61] Coen De Roover, Ralf Lämmel, and Ekaterina Pek. 2013. Multi-dimensional exploration of API usage. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 152–161. https://doi.org/10.1109/ICPC.2013.6613843

[62] Pasquale Salza, Fabio Palomba, Dario Di Nucci, Cosmo D'Uva, Andrea De Lucia, and Filomena Ferrucci. 2018. Do developers update third-party libraries in mobile apps?. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 255–265. https://doi.org/10.1145/3196321.3196341

[63] Anand Ashok Sawant and Alberto Bacchelli. 2015. A Dataset for API Usage. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, Massimiliano Di Penta, Martin Pinzger, and Romain Robbes (Eds.). IEEE Computer Society, 506–509. https://doi.org/10.1109/MSR.2015.75

[64] solid IT. 2019. *DB-Engines*. Retrieved June 27, 2019 from https://db-engines.com/en/ranking_osvsc

[65] Fabian Trautsch and Jens Grabowski. 2017. Are There Any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 207–218. https://doi.org/10.1109/ICST.2017.26

[66] Jason Tsay, Laura Dabbish, and James D. Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 356–366. https://doi.org/10.1145/2568225.2568315

[67] Pierre-Yves Vandenbussche, Ghislain Atemezing, María Poveda-Villalón, and Bernard Vatant. 2017. Linked Open Vocabularies (LOV): A gateway to reusable semantic vocabularies on the Web. *Semantic Web* 8, 3 (2017), 437–452. https://doi.org/10.3233/SW-160213

[68] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark G. J. van den Brand, Alexander Serebrenik, Premkumar T. Devanbu, and Vladimir Filkov. 2015. Gender and Tenure Diversity in GitHub Teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, Republic of Korea, April 18-23, 2015*, Bo Begole, Jinwoo Kim, Kori Inkpen, and Woontack Woo (Eds.). ACM, 3789–3798. https://doi.org/10.1145/2702123.2702549

[69] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. 2010. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference, 2010, Oxford, MS, USA, April 15-17, 2010*, H. Conrad Cunningham, Paul Ruth, and Nicholas A. Kraft (Eds.). ACM, 42. https://doi.org/10.1145/1900008.1900067

[70] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85. https://doi.org/10.1145/2629489

[71] Erik Wittern, Alan Cha, James C. Davis, Guillaume Baudart, and Louis Mandel. 2019. An Empirical Study of GraphQL Schemas. *arXiv e-prints*, Article arXiv:1907.13012 (Jul 2019), arXiv:1907.13012 pages. arXiv:cs.SE/1907.13012

[72] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60. https://doi.org/10.1145/2206869.2206879

[73] Tao Xie and Jian Pei. 2006. MAPO: mining API usages from open source repositories, See [20], 54–57. https://doi.org/10.1145/1137983.1137997

[74] Ahmed Zerouali and Tom Mens. 2017. Analyzing the evolution of testing library usage in open source Java projects, See [56], 417–421. https://doi.org/10.1109/SANER.2017.7884645

[75] Hao Zhong and Hong Mei. 2019. An Empirical Study on API Usages. *IEEE Trans. Software Eng.* 45, 4 (2019), 319–334.

[76] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science)*, Sophia Drossopoulou (Ed.), Vol. 5653. Springer, 318–343. https://doi.org/10.1007/978-3-642-03013-0_15