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
johannaharterel@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:

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.

\[
\text{SELECT } p, k \text{ WHERE }
\begin{align*}
  & \text{?p a foaf:Person ; foaf:knows ?k .} \\
  & \text{?k a foaf:Person}
\end{align*}
\]

\[
\text{MATCH (p:Person)-[:KNOWS]->(k:Person) RETURN p, k}
\]

\[
\begin{aligned}
  g . V ()
  & . hasLabel (‘person’) . as (‘p’) \\
  & . outE (‘knows’) . inV ()
  & . hasLabel (‘person’) . as (‘k’)
  & . select (‘p’, ‘k’)
\end{aligned}
\]

\[
\begin{aligned}
  \text{person} \\
  & \text{name} \\
  & \text{knows} \{ \text{name} \}
\end{aligned}
\]

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

\textbf{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.

\textbf{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.

\textbf{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.

\textbf{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.
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.

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.

<table>
<thead>
<tr>
<th>Language</th>
<th>Dependencies</th>
<th>Groups</th>
<th>Per Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARQL</td>
<td>191</td>
<td>72</td>
<td>2.65</td>
</tr>
<tr>
<td>Cypher</td>
<td>80</td>
<td>14</td>
<td>5.71</td>
</tr>
<tr>
<td>GraphQL</td>
<td>343</td>
<td>140</td>
<td>2.45</td>
</tr>
<tr>
<td>Gremlin</td>
<td>84</td>
<td>35</td>
<td>2.40</td>
</tr>
<tr>
<td>XQuery</td>
<td>24</td>
<td>18</td>
<td>1.33</td>
</tr>
<tr>
<td>SQL</td>
<td>3336</td>
<td>~2058</td>
<td>~1.62</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>System</th>
<th>Search Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maven</td>
<td>/code?q=$g+$a+in:file+filename:pom.xml</td>
</tr>
<tr>
<td>Gradle</td>
<td>/code?q=$g+$a+in:file+extension:gradle</td>
</tr>
<tr>
<td>Ant/Ivy</td>
<td>/code?q=$g+$a+in:file+filename:ivy.xml</td>
</tr>
<tr>
<td>Ant/Ivy</td>
<td>/code?q=$g+$a+in:file+filename:build.xml</td>
</tr>
</tbody>
</table>

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 $D_{total}$.

\(^2\)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 $D_{total}$ as $D_{popular}$.

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 $D_{queries}$.

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 (\(\ldots\)) 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 .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 ($D_{total}$) 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:


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 $D_{queries}$ (see 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.
We also apply additional popularity filters to reduce the 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 \(D_{total}\) 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 3.** Overview of our extended approach (related to RQ2) as an extension to Figure 2.
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 occurring 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 $D_{total}$ 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 4.** Venn diagram of repositories per language.
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).

In 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. 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.

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.

<table>
<thead>
<tr>
<th>dep. var: is_sparql</th>
<th>Model 0</th>
<th>Model 1</th>
<th>Model 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale(np.log(active_developers + 0.5))</td>
<td>0.134***</td>
<td>0.117***</td>
<td>0.06***</td>
</tr>
<tr>
<td>scale(np.log(active_files + 0.5))</td>
<td>0.043***</td>
<td>0.019**</td>
<td>0.017**</td>
</tr>
<tr>
<td>scale(np.log(created_days_ago))</td>
<td>0.038***</td>
<td>0.04***</td>
<td>0.04***</td>
</tr>
<tr>
<td>scale(np.log(stargazers_count + 0.5))</td>
<td>0.038***</td>
<td>0.04***</td>
<td>0.04***</td>
</tr>
<tr>
<td>R-squ.</td>
<td>0.017</td>
<td>0.089</td>
<td>0.102</td>
</tr>
</tbody>
</table>

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

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 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. 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 $D_{ranked}$, consisting of 150 repositories. We review the top ranking 75 (50%) manually. This review

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.

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.
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 example 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

<table>
<thead>
<tr>
<th>Repository Description</th>
<th>#</th>
<th>Domain</th>
<th>Specific Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content management for conservators.</td>
<td>1</td>
<td>Document Management</td>
<td>Conservation</td>
</tr>
<tr>
<td>Model conversion in context of grid simulation models.</td>
<td>2</td>
<td>Document Management</td>
<td>Library</td>
</tr>
<tr>
<td>Semantic web application framework (that uses SPARQL).</td>
<td>3</td>
<td>Document Management</td>
<td>Scholarly/Authorship</td>
</tr>
<tr>
<td>RDF-based ETL tool.</td>
<td>4</td>
<td>Process Management</td>
<td>ETL Process</td>
</tr>
<tr>
<td>Service for summarizing (specific) RDF graphs.</td>
<td>5</td>
<td>Process Management</td>
<td>Cloud Management</td>
</tr>
<tr>
<td>Meta-cloud managing network description model.</td>
<td>6</td>
<td>Process Management</td>
<td>Workflow</td>
</tr>
<tr>
<td>Visualization and interaction for linked, geospatial data.</td>
<td>7</td>
<td>Data Visualization</td>
<td>Summarization</td>
</tr>
<tr>
<td>Oslo public library management system.</td>
<td>8</td>
<td>Data Visualization</td>
<td>Geospatial Data</td>
</tr>
<tr>
<td>Text-based narrative generation for workflow results.</td>
<td>9</td>
<td>Data Visualization</td>
<td>Workflow Narrative</td>
</tr>
<tr>
<td>Management system of complex scholarly contribution relations.</td>
<td>10</td>
<td>Data Visualization</td>
<td>GWAS</td>
</tr>
<tr>
<td>Visualization component of GWAS catalog curation tool.</td>
<td>11</td>
<td>Data Visualization</td>
<td>Model Conversion</td>
</tr>
<tr>
<td>Web App Framework</td>
<td>12</td>
<td>Web Application</td>
<td>Web App Framework</td>
</tr>
</tbody>
</table>

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 CONSTRUCT without also using CONSTRUCT queries. Half of all projects use ASK queries in addition to SELECT (and other) query kinds.
We only include repositories that directly depend on libraries whether such ontologies exist, not the quantity. For domain specific or specialized ontologies, that relate to very specific domain related ontology.

Secondly, we do not select repositories that only transitively depend on these dependencies. In- stead, 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 Android 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 revisions 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

### 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

![Figure 7. Number of queries divided by query kind for the 12 applications labeled as concrete.](image-url)
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 predominantly 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


[73] Tao Xie and Jian Pei. 2006. MAPO: mining API usages from open source repositories, See [56], 417–421. https://doi.org/10.1109/2629489

