Automatic Query-centric API for Routine Access to Linked Data*

Albert Meroño-Peñuela[1] and Rinke Hoekstra[1,2]

[1] Computer Science Department, Vrije Universiteit Amsterdam, NL
{albert.merono,rinke.hoekstra}@vu.nl
[2] Faculty of Law, University of Amsterdam, NL

Abstract. Despite the advatages of Linked Data as a data integration paradigm, *accessing* and consuming Linked Data is still a cumbersome task. Linked Data applications need to use technologies such as RDF and SPARQL that, despite their expressive power, belong to the data *integration* stack. As a result, applications and data cannot be cleanly separated: SPARQL queries, endpoint addresses, namespaces, and URIs end up as part of the application code. Many publishers address these problems by building RESTful APIs around their Linked Data. However, this solution has two pitfalls: these APIs are costly to maintain; and they blackbox functionality by hiding the queries they use. In this paper we describe `grlc`, a gateway between Linked Data applications and the LOD cloud that offers a RESTful, reusable and uniform means to routinely access *any* Linked Data. It generates an OpenAPI compatible API by using parametrized queries shared on the Web. The resulting APIs require no coding, rely on low-cost external query storage and versioning services, contain abundant provenance information, and integrate access to different publishing paradigms into a single API. We evaluate `grlc` qualitatively, by describing its reported value by current users; and quantitatively, by measuring the added overhead at generating API specifications and answering to calls.

Keywords: Linked Data, API, REST, SPARQL, Data access, OpenAPI

1 Introduction

Data integration across multiple sources is an important challenge in the development of information systems [2]. The Linked Data [9] publishing paradigm is designed to make the Web evolve into a global *dataspace* [1] through syntaxes for data standardization and linkage, and query languages (RDF, SPARQL). These technologies have steep learning curves and limited adoption [24], which has resulted in a dataspace that is quite *heterogeneous* compared to, and *distinct* from, more mainstream Web-based architectures that use RESTful Application

*This paper is a significantly revised and extended version of a paper about an earlier version of `grlc` presented at the SALAD 2016 workshop [16].

Programming Interfaces (APIs) to mediate between the application and the underlying data. This heterogeneity is very tangible when we *access* Linked Data, which can be done in multiple ways: by submitting SPARQL queries to endpoints, downloading RDF dumps, parsing RDFa in HTML pages, or as Linked Data Fragments; to name a few. Different requirements drive the choice for each of these methods, but it is the *publisher* who is in control. Client applications need to be specifically tailored to each of these methods to consume Linked Data. This creates two problems. First, the data models easily become intertwined with application code, generating hard-coded queries that are difficult to maintain and share. Second, the disconnect with mainstream remote data access in Web development (which simply requires HTTP and JSON) undermines the adoption of Semantic Web technology, decreasing the "market value" of published Linked Data.

A number of solutions have been proposed to overcome these problems. Work such as [23] and the smartAPI [26] propose to expose REST APIs as Linked Data. Despite the value of this for clients with high expressivity requirements, these solutions pose an additional (API) integration problem for clients that need to query Linked Data without having to learn complex query languages, with low expressivity requirements, and in conjunction with other non-Semantic Web data sources. We aim at this specific target user community. Accordingly, we build from existing work targeting this community, such as the OpenPHACTS platform [6], LDtogo [18] and the BASIL server [3], which deploy APIs on top of their internal Semantic Web stacks, functioning as wrappers around their Linked Data endpoints. However, these solutions have two pitfalls. First, the APIs need to be routinely written and maintained by (costly) developers. Second, they typically blackbox the queries they use under the hood, offering a mutually exclusive solution of either using queries *or* API calls, but not both.

Publishing an API that simply executes Semantic Web queries *should be as easy as sharing these queries*. This is the basic idea of `grlc` [16], which clearly separates the workflows of *query maintenance* and *API construction*. As a result, it allows for a neat, open and collaborative management of queries (typically via GitHub repositories), and uses the logic of this management to build equivalent Linked Data APIs automatically on demand. In this paper, we extend `grlc` turning it into a generic Linked Data gateway that provides uniform API access to *any* Linked Data published in SPARQL endpoints, Linked Data Fragments servers, RDF dumps, or RDFa embedded in HTML pages. Its added values are the clear decoupling between different Linked Data access requirements; the zero-effort of coding APIs for accessing Linked Data; and the non-blackboxing of queries, which remain always available. Concretely, the contributions of this paper are:

- Architectural guidelines for decoupling semantic queries from application code (Section 2);
- A system architecture that generates OpenAPI specifications and enables API call name executions using remote Git repositories containing SPARQL, triple pattern fragments, dump, or RDFa queries (Sections 2, 2.3);

– Rich features that such guidelines and architecture enable, like zero-effort versioning and API provenance (Sections 2.5, 2.6)
– A qualitative evaluation (Section 3.1) providing evidence of use and fulfillment of requirements by users;
– A quantitative evaluation (Section 3.2) measuring `grlc`'s overhead.

2 System Architecture

`grlc`[3] is a lightweight middleware that automatically builds complete, well documented, and neatly organized Linked Data APIs on the fly in a query-centric way, effectively allowing client applications to access any Linked Data via RESTful APIs. This includes Linked Data exposed in SPARQL endpoints, but also in Linked Data Fragments servers [25], RDF dumps, and HTML pages enriched with RDFa (see Section 2.3). `grlc` provides three basic operations: (1) it generates the OpenAPI specification for the queries contained in a given repository; (2) it forwards browsers to the Swagger UI[4] to provide an interactive user-facing frontend of the API contents; and (3) it translates `http` requests to call the operations of the API against a SPARQL endpoint with several parameters. A docker bundle for easy deployment is available in Docker Hub via `docker pull clariah/grlc`

`grlc`'s system architecture is shown in Figure 1. The basic idea behind `grlc` is depicted at the bottom: an external *query provider* (typically GitHub or GitLab) is responsible for storing, versioning and exposing semantic queries via Git and `http`. This decouples these queries, and their curation workflows, from applications using them; and in particular, from applications generating APIs on top of them. The typical use cases start when a client application wants to generate an *OpenAPI spec* or *execute a call name*. When generating an OpenAPI spec, `grlc` retrieves metadata from the query provider (query names, descriptions, versions, endpoints, etc.) and uses them to build a valid API specification that mimics the organization of the query repository. To extract these metadata, `grlc` uses a *YAML parser* for enriched query decorators (see Section 2.1) and a *parameter parser* for mapping API parameters with query variables (see Section 2.2). When executing a call name, `grlc` retrieves the original content from the query provider, and uses the *query rewriter* to replace query variables with parameter values. Next, it sends the rewritten query to its corresponding endpoint, gets the results, and passes them on to the client application.

To run these workflows, `grlc` uses a simple API that allows client applications to express what APIs (call names) to generate (execute). Let us assume that our query provider is GitHub, and that we are using the public instance of `grlc` at `http://grlc.io/`.[5] If the GitHub repository containing queries is at `https://github.com/:owner/:repo`, then the `grlc` API provides the following routes:

[3] Source code at `https://github.com/CLARIAH/grlc`; public instance at `http://grlc.io/`.
[4] See `https://github.com/swagger-api/swagger-ui`
[5] These are the defaults, but can be customized in different configuration files.

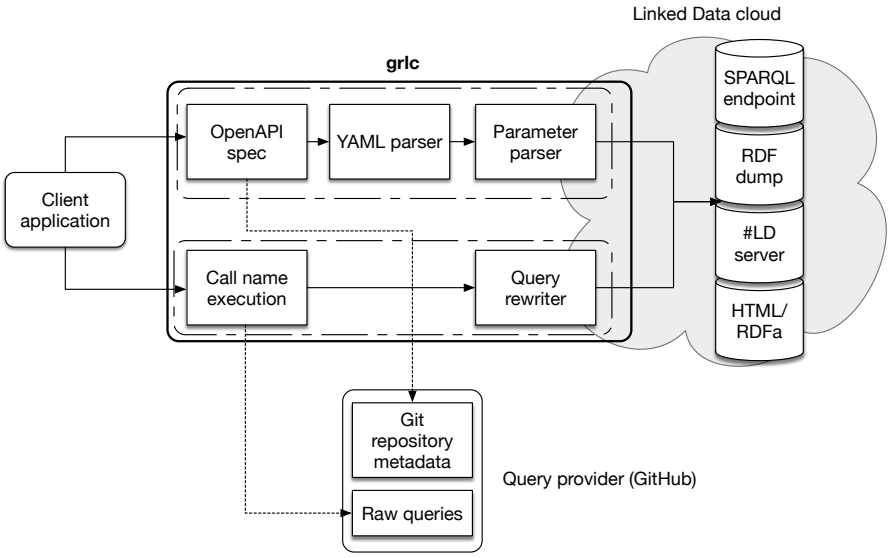


Fig. 1: Architecture of `grlc`. Linked Data sources and query providers are external to the system, and used to access and build Linked Data APIs.

- `http://grlc.io/api/:owner/:repo/spec`: JSON OpenAPI-compliant specification for the queries of `:owner` at `:repo`.
- `http://grlc.io/api/:owner/:repo/api-docs`: Swagger-UI, rendered using the previous JSON spec, as shown in Figure 1.
- `http://grlc.io/api/:owner/:repo/`: Same as previous.
- `http://grlc.io/api/:owner/:repo/:operation?p_1=v_1...p_n=v_n`: `http GET` request to `:operation` with parameters $p_1, ..., p_n$ taking values $v_1, ..., v_n$.

2.1 Query Decorators

To generate rich, accurate and descriptive OpenAPI specificaitons, we use *SPARQL decorators* to add metadata in queries at the query provider. These metadata do not pollute the query contents, since we implement them as comments before the query. Each query translates into an API operation. The syntax is depicted in the following example[6]:

```
#+ summary: A brief summary of what the query does
#+ method: GET
#+ endpoint: http://dbpedia.org/sparql
#+ tags:
#+     - I am a tag
#+     - Awesomeness
#+ enumerate:
```

[6] Additional examples can be found at `http://grlc.io`

```
#+     - var_1
#+     - var_2
#+ defaults:
#+     - var_1 : "foo"
#+ pagination: 100
```

These indicate the *summary* of the query (which will document the API operation), the `http` method to use (`GET`, `POST`, etc.), the *endpoint* to send the query to, and the *tags* under which the operation falls in. The latter helps to keep operations organized within the API. The decorator *enumerate* allows for generating the enumerations (possible values) of parameters for the specified variables; similarly, *defaults* allows specifying a default value for a parameter (see Section 2.2). The *pagination* value tells `grlc` to return the query results in pages of the indicated result size.

In addition, we suggest to include two special files in the repository. The first is a `LICENSE` file containing the license for the SPARQL queries and the API. The second is the `endpoint.txt` file, with the URI of a default endpoint to direct all queries of the repository; this allows for fast endpoint switching and enables an easier query reuse. `grlc` gives preference to the endpoint at the `#+ endpoint` decorator, then the `endpoint.txt` file, and finally a default one[7].

2.2 Parameter Mapping

It is often useful for SPARQL queries to be parameterized. This happens when a resource in a basic graph pattern (BGP) can take specific values that affect the result of the query. Previous work has investigated how to map these values to parameters provided by the API operations [3,6].

`grlc` follows BASIL's convention for mapping HTTP parameters to SPARQL[8], by interpreting some "parameter-declared" SPARQL variables as parameter placeholders. An example parametrized query[9] is shown in Listing 1.1. SPARQL variable names staring with `?_` and `?__` indicate mandatory and optional parameters. If they end with `_iri` or `_integer`, they are expected to be mapped to IRIs and literal (integer) values. API operations of the form `http://grlc.io/:owner/:repo/:operation?p_1=v_1...p_n=v_n` using these queries are executed as follows: `grlc` first retrieves the raw SPARQL query from the query provider (see Figure 1); and secondly it replaces the placeholders by the parameter values $v_1, ..., v_n$ supplied in the API request. After this, the query is submitted to the endpoint (see Secion 2.1) and results are forwarded to the client.

Parameter enumerations. To guide users at providing valid parameter values, `grlc` tries to fill the enumeration `get->parameters->enum` of the OpenAPI specification, which (optionally) lists avilable parameter values. To generate it, `grlc` sends an additional SPARQL query to the endpoint, using the original BGP but projecting all parameter variables to obtain their bindings. Figure 2 shows an example of how the Swagger UI displays parameter enumerations.

[7] In `http://grlc.io/` this is DBpedia's endpoint.

[8] See `https://goo.gl/K0YQDK`

[9] The original query can be found at `https://goo.gl/P5nvml`

```
SELECT (SUM(?pop) AS ?tot) FROM <urn:graph:cedar-mini:release> WHERE {
        ?obs a qb:Observation.
        ?obs sdmx-dimension:refArea ?_location_iri.
        ?obs cedarterms:Kom ?_kom_iri.
        ?obs cedarterms:population ?pop.
        ?slice a qb:Slice.
        ?slice qb:observation ?obs.
        ?slice sdmx-dimension:refPeriod ?_year_integer.
        ?obs sdmx-dimension:sex ?_sex_iri.
        ?obs cedarterms:residenceStatus ?_residenceStatus_iri.
        FILTER (NOT EXISTS {?obs cedarterms:isTotal ?total }) }
```

Listing 1.1: Example of a parametrized SPARQL query (prefixes ahave been omitted).

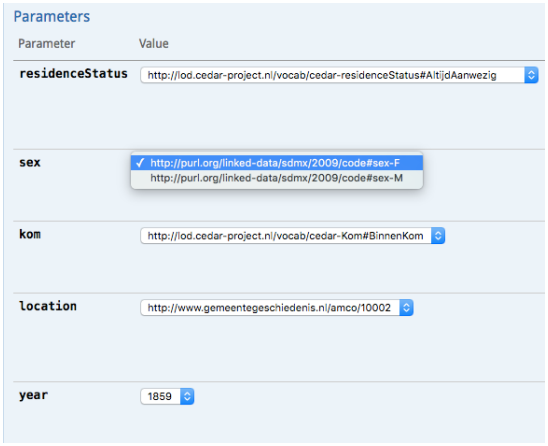


Fig. 2: Screenshot of the Swagger user interface rendering parameter enumerations generated by `grlc`.

2.3 Access to Any Linked Data

`grlc` acts as a multiplexer between the different Linked Data access methods. The currently supported access methods include SPARQL endpoints, Linked Data Fragments servers, RDF dumps, and HTML pages with RDFa markup. SPARQL queries are detected as files with the extensions `.rq` and `.sparql` in the remote repository (`.tpf` for triple pattern fragments). Queries against RDF dumps and HTML embedded RDFa are detected by the decorators `#+ mime: turtle`[10] and `#+ mime: rdfa`; in such cases the endpoint must point to RDF/RDFa resources. This provides three advantages for Linked Data consumers. First, it hides from them the specific Linked Data access method used by publishers, offering a universal Web API that operates over these methods and only demands

[10]The `xml`, `n3`, `nt`, `trix` and `nquads` syntaxes are also supported.

HTTP requests. Second, it integrates Linked Data sources independently of these publication methods into a standard Web API. And third, it allows for quickly and effectively switching the queries targets if needed.

2.4 Content Negotiation

`grlc` supports content negotiation at two different levels: by *request*, and by *URL*. By request, `grlc` checks the value of the `Accept` header in incoming `http` requests. By URL, `grlc` checks whether a route calling an API operation ends with a trailing `.csv`, `.json` or `.html`. In both cases, the corresponding `Accept` `http` header is used in the request to the SPARQL endpoint, delegating support of specific content types to each endpoint. When the response from the server is received, `grlc` sets the `Content-Type` header of the client response to match that received by the endpoint, and therefore it only proxies both requests and responses.

2.5 Commit-based APIs

Often, applications depend on specific versions of APIs and queries to function properly. `grlc` uses the underlying versioning logic of Git to generate API versions that match the different query versions. The default behavior is to use the contents of the HEAD pointer in the master branch of the query provider repository. In this case, `grlc`'s routes work as described in Section 2. Otherwise, the following routes use commit hashes to interact with the API of a commit-specific version of the queries:

- `http://grlc.io/api/:owner/:repo/commit/:sha/spec`: JSON OpenAPI-compliant specification for the queries of `:owner` at `:repo` with the commit hash `:sha`.
- `http://grlc.io/api/:owner/:repo/commit/:sha/api-docs`: Swagger-UI for the commit hash `:sha`, rendered using the previous JSON spec, as shown in Figure 1.
- `http://grlc.io/api/:owner/:repo/commit/:sha/`: Same as previous.
- `http://grlc.io/api/:owner/:repo/commit/:sha/:operation?p_1=v_1...p_n=v_n`: `http` `GET` request to commit hash `:sha` of `:operation` with parameters $p_1, ..., p_n$ taking values $v_1, ..., v_n$.

In these cases, the OpenAPI specification will be generated on the basis of what that specific commit contains; calls to commit-specific operations work likewise. To ease user interaction and browsing across versions, we add links to the generated OpenAPI spec and Swagger-UI to the next and previous versions (i.e. commit), if available. All APIs generated by `grlc` are versioned using their corresponding commit hashes.

2.6 Provenance

One advantage of `grlc` over other Linked Data API methods is that it does not use APIs to blackbox queries, allowing both queries and APIs to be used simultaneously. To further enhance its transparency and explainability, `grlc` generates

provenance using the W3C PROV [7] standard in two different ways. First, it generates a graph representing the workflow of creating the OpenAPI specifcation by reusing externally retrieved queries. Second, it adds to this graph the PROV representation of the Git history behind all queries reused, by calling Git2PROV [4]. To allow the exploration of all this provenance information, we integrate the visualizations of PROV-O-Viz [10], accessible via an *Oh yeah?* button in the Swagger-UI page of the API specification.

2.7 SPARQL2Git

Interacting with languages like SPARQL and technologies like Git can be tedious for some users. To alleviate this, `grlc` works in conjunction with another tool: SPARQL2Git[11] [17]. SPARQL2Git combines a user interface for comfortably editing and trying SPARQL queries and their decorators (see Section 2.1), with a transparent use of the GitHub API. Users can "save" versions of their queries and the system deals with managing commits on their Git repositories. A `grlc` button is always accessible to try out the APIs generated from their committed queries.

3 Evaluation

We evaluate `grlc` in two different ways: qualitatively, and quantitatively. In the qualitative evaluation, we provide testimonies of the utility of `grlc` for (third party) organizations and projects. In the quantitative evaluation, we study the performance of `grlc`. First, we investigate its overhead over direct SPARQL queries (Section 3.2). Secondly, we benchmark the speed in which it generates OpenAPI specifications (Section 3.2).

3.1 Qualitative Evaluation

From the start of its operation in July 2016, the public instance of `grlc` has attracted 646 unique visitors, 46.4% of return rate, and generating 1,205 sessions. `grlc` has also attracted the attention of external developers, who have sent 13 pull requests that have been integrated into the master branch. A list of community maintained queries and matching APIs is available at `http://grlc.io`.

In this section we evaluate the requirements satisfied by `grlc` in a number of external institutions in 6 different domains where `grlc` is being currently used. We asked members of these institutions to describe their use cases, the advantages and disadvantages of addressing them by using `grlc`, and their motivation for choosing it over other solutions.

[11] See `http://sparql2git.com`

DANS: Historical Statistics The Netherlands Institute for Permanent Access to Digital Research Resources (DANS) publishes the Dutch historical censuses (1795–1971) as Linked Data [15].[12]. Queries across this data are maintained on GitHub. These queries are used across various client applications,[13] and other organizations (Statistics Netherlands, a.o.) inspiring the need for a shared API. The then existing lightweight solutions, such as BASIL[14] and implementations of the Linked Data API created a maintenance problem as they require one to keep multiple copies of the same queries in different places. Given the frequency of mutations in the queries, this was problematic. The `grlc` system allows queries to be maintained in a single location, and offers an ecosystem where SPARQL and non-SPARQL savvy applications coexist.

IISH: Social History The International Institute for Social History partners in the CLARIAH[15] project for digital humanities. Typical social history research requires querying across combined, structured humanities data, and performing statistical analysis in e.g. `R` [11]. Given that there are potentially infinitely many such research queries, building a one-size-fits all API is not feasible. The `R` SPARQL package [8] allows one to use SPARQL queries directly from `R`. However, this results in hard-coded, non reusable, and difficult to maintain queries. As shown in Figure 3, with `grlc` the `R` code becomes *clearer* due to the decoupling with SPARQL; and *shorter*, since a `curl` one-liner calling a `grlc` enabled API operation suffices to retrieve the data. Furthermore, the exact query feeding the research results can be stored, and shared with fellow scholars and in papers.

National eScience Center: Cultural Heritage The National eScience Center uses `grlc` in a tool for Linked Data exploration of cultural heritage data (Dive+). The Dive+ UI calls the `grlc`-generated API to access underlying data. The `grlc` code is included as a library to augment parts of the Dive+ API that are not Linked Data data-access related (e.g. search, legacy data). The advantage of using `grlc` is that it allows NLeSC to manage SPARQL queries separate from the rest of the API – this enables, for instance, to have different queries without having to deploy a new version of the API. NLeSC used `grlc` instead of other solutions because it was easy to deploy and open source.

TNO: FoodCube The Netherlands Organisation for Applied Scientific Research (TNO) uses `grlc` in a food related project for the municipality of Almere. FoodCube aims to provide an integrated view to all kinds of datasets related to the food supply chain; domain knowledge and interesting domain questions are the core focus. FoodCube uses `grlc` to provide 'FAQ' (Frequently Asked SPARQL

[12] This was done through the CEDAR project, see `http://www.cedar-project.nl/` and `https://github.com/CEDAR-project/Queries`

[13] YASGUI-based browsing: `http://lod.cedar-project.nl/cedar/data.html`, drawing historical maps with census data: `http://lod.cedar-project.nl/maps/map_CEDAR_women_1899.html`

[14] `https://github.com/the-open-university/BASIL`

[15] `http://clariah.nl/`

Questions) for those who would prefer REST over SPARQL, but also to explore the data. This is made possible by the ability to annotate the SPARQL queries with keywords and a description.

NewGen Chennai: Conference Proceedings NewGen uses `grlc` to build the IOS Press ECAI API. Their goal is to expose the ECAI conference proceedings not only as Linked Data that can be used by Semantic Web practitioners, but also as a Web API that web developers can consume. This is useful for bringing together and bridging the two communities and rich ecosystems of software. Key features of `grlc` for this use case are query curation, sharing and dissemination. For this last point, being able to provide metadata to individual queries is reportedly very useful. NewGen finds easy to use and document the API, and to set-up. Similarly, the use of Git as a backend is an advantage, and they consider the `grlc` development community helpful. SPARQL2Git (see Section 2.7) emerged as a requirement for a query curation frontend. Other alternatives[16] were considered, but the two advantages of `grlc` were its use of GitHub for ingesting community curated queries, and the minimum infrastructure/resources needed for building APIs.

EU RISIS: Science, Technology and Innovation `grlc` is currently used within the Semantically Mapping Science (SMS) platform[17] for sharing of SPARQL queries and thereby their results among multiple researchers. As technical core within the RISIS EU project[18], SMS aims to provide a data integration platform where researchers from science, technology and innovation (STI) can find answers to their research questions. The SMS platform provides a faceted data browser where interactions of non-linked-data expert users are translated into a set of complex SPARQL queries, which are then run to aggregate data from relevant SPARQL endpoints. One of the challenges within the platform was how to share, extend and repurpose user-generated queries in a flexible way. `grlc` helps to address this issue by providing a URI for the resulted queries and by supporting collaborative update of those queries. Furthermore, creating Linked Data APIs on top of `grlc` enables external applications to reuse and exploit some of the features of the SMS platform, e.g. SMS geo-services to annotate addresses within a spreadsheet document.

3.2 Quantitative

Call Execution Overhead Here, we quantify the added overhead of `grlc` as a middleware between Web clients and Linked Data providers. To do so, we compare the execution times of sending SPARQL queries directly to a SPARQL

[16] Reportedly `https://github.com/danistrebel/SemanticGraphQL`, `https://github.com/nelson-ai/semantic-graphql` and `https://github.com/ColinMaudry/sparql-router/wiki/Using-SPARQL-router`.

[17] `http://sms.risis.eu`

[18] `http://risis.eu`

```
## using grlc API call
library(RCurl)
canada <- getURL("http://grlc.clariah-sdh.eculture.labs.vu.nl/clariah/wp4-
canada <- read.csv(textConnection(canada))
sweden <- getURL("http://grlc.clariah-sdh.eculture.labs.vu.nl/clariah/wp4-
sweden <- read.csv(textConnection(sweden))

fit_canada_base <- lm(log(hiscam) ~ log(gdppc), data=canada)
fit_canada <- lm(log(hiscam) ~ log(gdppc) + I(age^2) + age, data=canada)
fit_sweden_base <- lm(log(hiscam) ~ log(gdppc), data=sweden)
fit_sweden <- lm(log(hiscam) ~ log(gdppc) + I(age^2) + age, data=sweden)
```

Fig. 3: The use of `grlc` makes Linked Data accessible from any `http` compatible application.

endpoint over HTTP, and calling the equivalent service names containing such SPARQL queries using `grlc`.

We use the SPARQL queries of the SP^{2}Bench SPARQL Performance Benchmark[19] (SP2B) [22]. All runs in one single node inside a `lxc` container running Linux Ubuntu 14.04.4 LTS, an Intel(R) Xeon(R) E5645 CPU at 2.40GHz, and 98GB of memory. As a backend triplestore we use a Virtuoso Open Source Edition 6.1.6. To avoid the influence of network traffic on our tests, we configure `grlc` to use local namespaces to resolve API calls and dereferencing query contents.[20] To make comparisons fair, both systems are queried using `curl`, making HTTP GET requests, and requesting the query results as CSV by setting the HTTP header `Accept: text/csv`. We disable all `grlc`'s caching mechanisms.[21]

Figure 4 shows the results of executing the SP2B queries on datasets of 50K, 500K, and 5M triples, sending HTTP requests to (a) directly to the SPARQL endpoint (submitting the query as a parameter of the request); and (b) using `grlc` (calling the equivalent call name in a Linked Data API generated using such queries[22]). We observe that, in queries that Virtuoso takes a considerable time (above 100ms), `grlc` only adds a marginal overhead (e.g. *q2*, *q7*, *q9*, *11*); contrarily, the impact of `grlc` is higher in fast queries below that threshold. We calculate the *relative overhead* of `grlc` with $\frac{t_g - t_v}{t_g}$, where t_g is the time consumed by `grlc`, and t_v is the time consumed by the SPARQL endpoint. Figure 5a shows the dependency of this ratio with the total execution time t_g. We observe that, for queries that SPARQL endpoints can solve very quickly (e.g. less than 200ms), more than 50% of the time is spent in `grlc` rather than at the endpoint. In even faster queries (e.g. below 100ms), the ratio taken by `grlc` is even larger (above 75% of the time). Nonetheless, in queries that take more than 400ms `grlc`'s impact is more limited (less than 25%).

[19] See `http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B`

[20] For this, we implement a basic GitHub-like API, see `https://github.com/albertmeronyo/dummyhub`

[21] All measurements in this section apply to the first execfution only; subsequent executions are immediate due to caching.

[22] See `https://github.com/albertmeronyo/sp2b-queries` and `http://grlc.io/api/albertmeronyo/sp2b-queries`

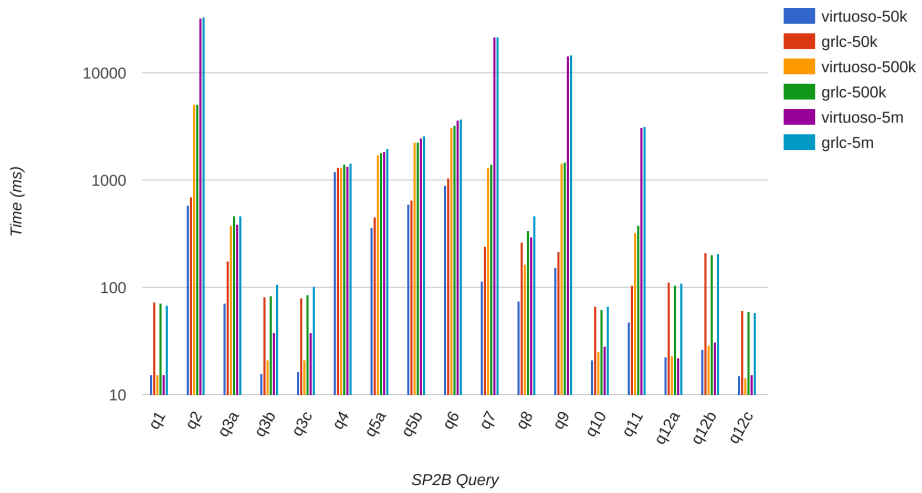
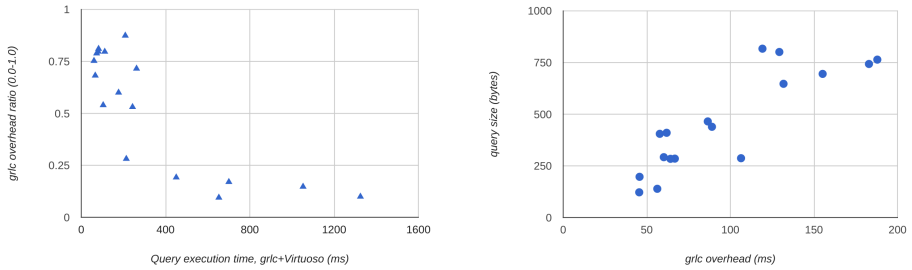


Fig. 4: Execution time of SP2B queries on 50K, 500K and 5M triple datasets, using: (a) Virtuoso alone; and (b) an instance of `grlc` that exposes the same queries as an API.

The *absolute overhead* of `grlc` is given by $t_g - t_v$, and equals on average over all queries 96.86 ± 46.83, 77.18 ± 46.48, and 80.87 ± 48.14 for the three dataset sizes. We observe here that, as expected, the cost of `grlc` is independent of the dataset size. However, there are some fluctuations in this cost that make it non-constant. A cause for the variability in `grlc`'s absolute cost can be observed in Figure 5b, which shows a linear relationship between `grlc`'s absolute overhead with respect to the size of the SPARQL queries. In summary, the penalties of `grlc`'s additional HTTP requests (needed for retrieving the query contents, the endpoint's URI, etc.) and their payloads are important contributors to its cost. In our tests, this cost is never higher than 187.9ms.

OpenAPI Specification Construction Here, we evaluate the cost of generating OpenAPI specifications with `grlc`. We use the same `grlc` instance (i.e. local API and query resolution) as described in the previous experiment (Section 3.2).

We create various OpenAPI specifications of different sizes and types. Spec sizes are determined by the number of call names (i.e. queries) contained in the spec, and we generate specs of 1, 10, 100 and 1000 call names. Query types are determined by the features typed in the query: we generate *plain* queries, containing only the query itself; *decorators* queries, also containing YAML metadata (endpoint URI, query summary, HTTP method, pagination, tags); and *enum*, also containing enumerated parameters. Figure 6 shows the time `grlc` spends on creating these specifications. We observe that in all cases this cost is linear with respect to the spec sizes (the time axis is in log scale). For APIs of conventional



(a) Execution time of a callname in `grlc` (x-axis), and share of this time taken by `grlc` (y-axis).

(b) `grlc` absolute overhead (y-axis) depending on the total size of a callname's query (x-axis).

Fig. 5: Breakdown of `grlc`'s overhead, and its dependency with total execution times and query size.

size (i.e. between 10 and 100 call names) containing only plain queries, `grlc` can generate specs between 335.4 ± 12.43 and $3,026.7 \pm 41.28$ms. The cost of adding useful decorators is only relatively more expensive for small APIs of 10 call names (510.8 ± 27.70ms), converging to the cost of plain queries ($3,388.5 \pm 27.72$ms) for larger 100 call name APIs. APIs containing many enumerated parameters are very expensive to generate ($34,658.7 \pm 70.21$ms for 10 call names), but single queries are more affordable ($3,487 \pm 18.32$ms).

4 Related Work

Decoupling Linked Data queries from the applications that use them follows principles of encapsulation and abstraction. There is abundant work in so-called SPARQL query repositories, which are fundamental to study the efficiency and reusability of methods querying Linked Data. SPARQL query logs, for instance, have been used to study differences between queries by humans and machines [20], and to understand how queried entities are semantically related [12]. Saleem et al. [21] propose to "create a Linked Dataset describing the SPARQL queries issued to various public SPARQL endpoints". Loizou et al. [14] identify (combinations of) SPARQL constructs that constitute a performance hit, and formulate heuristics for writing optimized queries.

The Semantic Web has developed a large body of work on the relationship between Linked Data and Web Services [5,19]. In [23], authors propose to expose REST APIs as Linked Data. These approaches suggest the use of Linked Data technology on top of Web services. Recently, the smartAPI [26] has proposed API building blocks for clients with with high expressivity requirements. Our work is related to results in the opposite direction, concretely the Linked Data API specification[23] and the W3C Linked Data Platform 1.0 specification,

[23] `https://github.com/UKGovLD/linked-data-api`

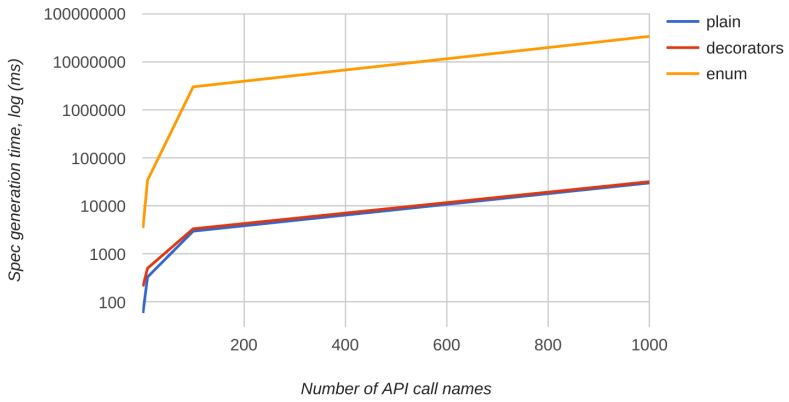


Fig. 6: Performance of `grlc` at creating OpenAPI specifications of different sizes and types.

which "describes the use of HTTP for accessing, updating, creating and deleting resources from servers that expose their resources as Linked Data"[24]. Kopecký et al. [13] address the specific issue of writing (updating, creating, deleting) these Linked Data resources via Web APIs. However, our work is more related to providing APIs that facilitate Linked Data access to a variety of publishing mechanisms. SPARQL is the most popular among such supported mechanisms in the OpenPHACTS Discovery Platform for pharmacological data [6], LDtogo [18] and the BASIL server [3]. These approaches build Linked Data APIs compliant with the Swagger RESTful API specification[25] that function as wrappers around SPARQL endpoints. Inspired by this, our work contributes additional: *(a)* decoupling with respect to the query storage and maintenance infrastructure, which we outsource to code repository providers; *(b)* abstraction over various Linked Data access methods (Linked Data Fragments, RDF dumps, HTML+RDFa) besides SPARQL; and *(c)* tools for automatically building well-documented API specifications.

5 Conclusions and Future Work

We have presented `grlc`, an automatic and query-centric method for enabling routine access to any Linked Data. `grlc` leverages the decoupling of semantic queries from applications, allowing query-based and API-based access simultaneously. It generates uniform and universal Web APIs irrespective of the Linked Data publishing method, making these Linked Data consumable and accessible to the mainstream Web community. It uses Git features to transparently

[24] `https://www.w3.org/TR/2015/REC-ldp-20150226/`

[25] `https://github.com/OAI/OpenAPI-Specification`

provide versioning and provenance. In the future, we plan on extending this work in multiple ways. First, we will enlarge our current supported infrastructures (GitHub, GitLab, SPARQL, dumps, etc.) to cover increasing requirements demanded by users. Secondly, we devise a JSON transformation language for customizing the structure of API results. Finally, we intend to investigate the reusability, exchangeability, and linkability of semantic query catalogs created by users of `grlc`.

Acknowledgements. This work was funded by the CLARIAH project of the Dutch Science Foundation (NWO). We want to thank all external users and contributors to this work, especially Carlos Marínez Ortiz, Ali Khalili, Barry Nouwt, and Trevor Lazarus. We also want to thank Laurens Rietveld for his technical suggestions, and Richard Zijdeman and Auke Rijpma for their testing.

References

1. A. Y. Halevy, M. J. Franklin, D.M.: Principles of dataspace systems. In: Proceedings of the Twenty-Fifth Symposium on Principles of Database Systems (PODS 2006). pp. 1—-9. ACM (2006)
2. Bernstein, P.A., Haas, L.M.: Information integration in the enterprise. Communications of the ACM 51(9), 72—79 (2008)
3. Daga, E., Panziera, L., Pedrinaci, C.: A BASILar Approach for Building Web APIs on top of SPARQL Endpoints. In: Services and Applications over Linked APIs and Data – SALAD2015 (ISWC 2015). vol. 1359. CEUR Workshop Proceedings (2015), http://ceur-ws.org/Vol-1359/
4. De Nies, T., Magliacane, S., Verborgh, R., Coppens, S., Groth, P., Mannens, E., Van de Walle, R.: Git2PROV: Exposing version control system content as W3C PROV. In: Poster and Demo Proceedings of the 12th International Semantic Web Conference (Oct 2013), http://www.iswc2013.semanticweb.org/sites/default/files/iswc_demo_32_0.pdf
5. Fielding, R.T.: Architectural styles and the design of network-based software architectures (2000)
6. Groth, P., Loizou, A., Gray, A.J., Goble, C., Harland, L., Pettifer, S.: API-centric Linked Data integration: The Open PHACTS Discovery Platform case study. Web Semantics: Science, Services and Agents on the World Wide Web 29(0), 12 – 18 (2014), http://www.sciencedirect.com/science/article/pii/S1570826814000195, life Science and e-Science
7. Groth, P., Moreau, L.: PROV-Overview. An Overview of the PROV Family of Documents. Tech. rep., World Wide Web Consortium (W3C) (2013), http://www.w3.org/TR/prov-overview/
8. van Hage, W.R., with contributions from: Tomi Kauppinen, Graeler, B., Davis, C., Hoeksema, J., Ruttenberg, A., Bahls., D.: SPARQL: SPARQL client (2013), http://CRAN.R-project.org/package=SPARQL, R package version 1.15
9. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space. Morgan and Claypool, 1st edn. (2011)
10. Hoekstra, R., Groth, P.: PROV-O-Viz - Understanding the Role of Activities in Provenance. In: 5th International Provenance and Annotation Workshop (IPAW 2014). LNCS, Springer-Verlag, Berlin, Heidelberg (2014)
11. Hoekstra, R., Meroño-Peñuela, A., Dentler, K., Rijpma, A., Zijdeman, R., Zandhuis, I.: An Ecosystem for Linked Humanities Data. In: Proceedings of the 1st Workshop on Humanities in the Semantic Web (WHiSe 2016), ESWC 2016 (2016)

12. Huelss, J., Paulheim, H.: The Semantic Web: ESWC 2015 Satellite Events, chap. What SPARQL Query Logs Tell and Do Not Tell About Semantic Relatedness in LOD, pp. 297–308. Springer International Publishing, Cham (2015), http://dx.doi.org/10.1007/978-3-319-25639-9_44
13. Kopecký, J., Pedrinaci, C., Duke, A.: Restful write-oriented api for hyperdata in custom rdf knowledge bases. In: Next Generation Web Services Practices (NWeSP), 2011 7th International Conference on. pp. 199–204 (Oct 2011)
14. Loizou, A., Angles, R., Groth, P.: On the formulation of performant {SPARQL} queries. Web Semantics: Science, Services and Agents on the World Wide Web 31, 1 – 26 (2015), http://www.sciencedirect.com/science/article/pii/S1570826814001061
15. Meroño-Peñuela, A., Guéret, C., Ashkpour, A., Schlobach, S.: CEDAR: The Dutch Historical Censuses as Linked Open Data. Semantic Web – Interoperability, Usability, Applicability (2015), in press
16. Meroño-Peñuela, A., Hoekstra, R.: grlc Makes GitHub Taste Like Linked Data APIs. In: The Semantic Web: ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May 29 – June 2, 2016. pp. 342–353. Springer (2016)
17. Meroño-Peñuela, A., Hoekstra, R.: SPARQL2Git: Transparent SPARQL and Linked Data API Curation via Git. In: Proceedings of the 14th Extended Semantic Web Conference (ESWC 2017), Poster and Demo Track. Portoroz, Slovenia, May 28th – June 1st. Springer (2017), (in print)
18. Ockeloen, N., de Boer, V., Aroyo, L.: LDtogo: A Data Querying and Mapping Frameworkfor Linked Data Applications. In: The Semantic Web: ESWC 2013 Satellite Events, Montpellier, France, May 26-30, 2013. pp. 199–203. Springer (2013)
19. Pedrinaci, C., Domingue, J.: Toward the next wave of services: Linked Services for the Web of data. Journ. of Universal Computer Science 16(13), 1694—1719 (2010)
20. Rietveld, L., Hoekstra, R.: Man vs. Machine: Differences in SPARQL Queries. In: Proceedings of the 4th USEWOD Workshop on Usage Analysis and the Web of of Data, ESWC 2014 (2014), http://usewod.org/files/workshops/2014/papers/rietveld_hoekstra_usewod2014.pdf
21. Saleem, M., Ali, M.I., Mehmood, Q., Hogan, A., Ngomo, A.C.N.: LSQ: Linked SPARQL Queries Dataset. In: The Semantic Web - ISWC 2015. LNCS, vol. 9367, pp. 261–269. Springer
22. Schmidt, M., Hornung, T., Küchlin, N., Lausen, G., Pinkel, C.: An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In: The Semantic Web - ISWC 2008: 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings. pp. 82–97. Springer (2008)
23. Speiser, S., Harth, A.: Integrating linked data and services with linked data services. In: The Semantic Web: Research and Applications. pp. 170—184. Springer (2011)
24. Vandenbussche, P.Y., Aranda, C.B., Hogan, A., Umbrich, J.: Monitoring the Status of SPARQL Endpoints. In: Proceedings of the ISWC 2013 Posters and Demonstrations Track, 12th International Semantic Web Conference (ISWC 2013). pp. 81–84. CEUR-WS (2013)
25. Verborgh, R., Hartig, O., Meester, B.D., Haesendonck, G., de Vocht, L., Sande, M.V., Cyganiak, R., Colpaert, P., Mannens, E., van de Walle, R.: Querying Datasets on the Web with High Availability. In: Proceedings of the 13th International Semantic Web Conference, ISWC2014 (2014)
26. Zaveri, A., Dastgheib, S., Whetzel, T., Verborgh, R., Avillach, P., Korodi, G., Terryn, R., Jagodnik, K., Assis, P., Wu, C., Dumontier, M.: smartAPI: Towards a more intelligent network of Web APIs (2017), (in print)