Knora Query Language (KnarQL)

Tobias Schweizer and Benjamin Geer

Digital Humanities Lab
University of Basel

9th May 2018

Abstract

SPARQL endpoints are one way to enable client software to query RDF data stored on a server, but they present a number of drawbacks that make them unsuitable for many applications. The Knora Query Language (KnarQL) is a dialect of SPARQL that aims to provide the power and flexibility of a SPARQL endpoint, while avoiding these problems and offering the advantages of a RESTful API. Its design suggests a practical way to go beyond some limitations of the ways that RDF data has generally been made available.

1 Introduction

SPARQL endpoints are one way to enable client software to query data stored in an RDF triplestore, but they present a number of drawbacks that make them unsuitable for many applications: there is no way to ensure that clients see only the data that they have permission to see, there is no provision for the versioning of data, and the ability to request query results of unlimited size poses scalability problems. The Knora Query Language (KnarQL), which uses a subset of SPARQL syntax, aims to provide the power and flexibility of a SPARQL endpoint, while avoiding these problems and offering the advantages of a RESTful API. It was implemented for use within a particular RDF application, but its basic design is of broader relevance, because it suggests a practical way to go beyond some limitations of the ways that RDF data has generally been made available.

In this design, the KnarQL query received from the client is not processed directly by the triplestore, but rather by a server application, which

translates KnarQL queries into SPARQL queries. This makes it possible for the ontologies used in the client's query to be simpler than the ones used in the triplestore, and allows the application to provide additional features not available in SPARQL, such as filtering results according to user permissions, enforcing the paging of results to improve scalability, taking into account the versioning of data in the triplestore, and returning responses in a form that is more convenient for web application development.

KnarQL has been developed as part of Knora (Knowledge Organization, Representation, and Annotation), a framework developed by the Data and Service Center for the Humanities (DaSCH) [1] that ensures the long-term availability and reusability of research data in the humanities. Knora is based on an RDF triplestore and a base ontology that can be further extended by project-specific ontologies. The Knora API server provides a RESTful API that allows data to be queried and updated, and supports the creation of virtual research environments capable of working with heterogeneous research data from different disciplines.

Knora's API not only facilitates accessing data in the triplestore by means of simple HTTP requests, but also provides authentication and authorisation as well as the versioning of data. KnarQL arose from the need for Knora to provide a powerful way of querying data, comparable to a SPARQL endpoint. Providing a flexible query language would reduce the number of specific use cases that would need their own API routes. However, it was apparent that a SPARQL endpoint would be incompatible with some of Knora's requirements: query results must be filtered according to the user's permissions, only the current versions of resources should be returned by default, and results must always be paged for scalability.

We initially considered creating a domain-specific language (DSL) for this purpose, but decided instead to adapt SPARQL, leveraging its standardisation and library support, while integrating it into Knora's API. KnarQL's syntax is thus a subset of the standard SPARQL CONSTRUCT query syntax, and it imposes some additional requirements on how queries must be written. Instead of simply returning a set of triples, a KnarQL query results in a JSON-LD response whose structure is meant to facilitate web application development.

Moreover, the ontologies that Knora uses to store data in the triplestore are based on design considerations that make them unsuitable for use in client queries. For example, in Knora, if there is an RDF resource of class `example:Person` with a property `example:name`, that property must be an `owl:ObjectProperty`, whose object is a `knora-base:TextValue` containing permissions, the value's creation date, a pointer to the previous version of the value, and so on, in addition to the person's name. A client that wants

to edit this value will need to know its IRI, and understand that the value is a complex object. But a client that simply wants to query the name would likely prefer to work with a simple `xsd:string`, both in the query language and in the results. This would also make it possible to represent `example:name` as a subproperty of a standard property such as `foaf:name`, whose object is a simple string.

Knora's solution to this problem, which is used in KnarQL, is to support different ontology 'schemas', which are like different views on a given ontology. Knora stores RDF data in the triplestore using its 'internal' schema, which includes permissions and versioning, as well as implementation details that clients should not have to deal with. This schema is not exposed on the Knora API. Instead, two API schemas are currently offered. A client that needs to update data uses the 'complex' schema, in which all values have IRIs. A client that simply queries data can use the 'simple' schema, in which values have simple datatypes such as `xsd:string`, thus making it possible to use standard properties such as `foaf:name`. Only the internal schema is used in the triplestore; the Knora API server converts data and ontology entities to and from the other schemas on the fly during request processing.

In this article, after an overview of existing solutions and developments (section 2), we present the general concept of KnarQL (sections 3 and 4) and an example of how it is being used in a digital edition project in the humanities (section 5).

2 Related Work

One way of making RDF data publicly available and queryable is by means of a SPARQL endpoint. Two prominent examples are DBpedia (`http://dbpedia.org/sparql`) and Europeana (`http://sparql.europeana.eu`). While SPARQL endpoints offer great flexibility and allow for complex queries, they have also been criticised. In a widely cited blog post, 'The Enduring Myth of the SPARQL Endpoint' (`https://daverog.wordpress.com/2013/06/04/the-enduring-myth-of-the-sparql-endpoint`), Dave Rogers argues that SPARQL endpoints are an inherently poor design that cannot possibly scale, and that RESTful APIs should be used instead.

GraphQL (`https://www.howtographql.com`) is a newer development and – despite its name – not restricted to graph databases. It is meant to be a query language that integrates different API endpoints. Instead of making several requests to different APIs and processing the results individually, GraphQL is intended to allow the client to make a single request that defines the structure of the expected response.

HyperGraphQL (`http://hypergraphql.org`), an extension to GraphQL, makes it possible to query SPARQL endpoints using GraphQL queries, by converting them to SPARQL. Its intended advantages include the reduction of complexity on the client side and a more controlled way of accessing a SPARQL endpoint, avoiding some of the problems discussed in Dave Rogers's blog post (`https://medium.com/@sklarman/querying-linked-data-with-graphql-959e28aa8013`). However, HyperGraphQL is designed to communicate directly with a SPARQL endpoint, and thus shares some of the limitations of SPARQL endpoints.

From our perspective, SPARQL endpoints and HyperGraphQL both have several drawbacks that make them unsuitable for Knora. First, they are based on the assumption that everything in the triplestore should be accessible to the client, and thus offer no way to restrict query results according to the client's permissions. Second, they assume that the data structures in the triplestore are the same as the ones to be returned to the client. Third, they do not enforce the paging of results, but leave this to the client. Fourth, they provide no way to work with data that has a version history (so that ordinary queries return only the latest version of each item). All these features are requirements for the DaSCH.

3 A Hybrid between a SPARQL Endpoint and a RESTful API

KnarQL is a hybrid between a SPARQL endpoint and a RESTful API, aimed at combining the advantages of both. A KnarQL query is a syntactically valid SPARQL CONSTRUCT query, but it is processed by a RESTful API server, which translates it into one or more SPARQL queries that are executed by the triplestore, and processes and transforms the results from the triplestore to construct the response sent back to the client. This extra layer of processing enables KnarQL to avoid the drawbacks of SPARQL endpoints.

3.1 Ontology Schemas

A design goal of KnarQL is to enable queries to work with data structures that are simpler than the ones actually used in the triplestore. To make this possible, Knora implements *ontology schemas*. Each ontology schema provides a different view on ontologies and data that exist in the triplestore. We use the term *internal schema* to mean the structures that are actually in the triplestore, and *external schema* to mean a view that transforms these structures in some way for use in the Knora API.

Knora requires each research project to define its own ontology or ontologies for the data it wishes to store. These ontologies must conform to the Knora data model and extend abstract entities provided in Knora's base ontology. Project-specific ontologies may also extend or refer to other ontologies, such as standard ones, that do not necessarily conform to the Knora data model and are not stored in the triplestore in Knora.

For each entity defined in an ontology in the triplestore (whether in a built-in Knora ontology or in a project-specific ontology), there is a corresponding entity in each external schema. The IRI of an internal entity is related to the IRI of the corresponding external entities according to simple formal rules. This makes it a simple matter to determine, from the form of a Knora entity IRI, which schema it belongs to.

Knora currently supports two external schemas, a *simple schema* that is designed for read-only operations such as KnarQL queries as well as for interoperability with standard ontologies, and a *complex schema* that is designed for editing (while still being simpler than the internal schema). Additional external schemas could be added in future. The Knora API server converts ontology entities and data between schemas on the fly; the triplestore deals only with the internal schema.

Because the complex schema is designed to enable data to be edited via a RESTful API, each editable value has an IRI. This contrasts with the design of standard ontologies, in which values are typically literals. However, in the simple schema, values are represented as literals, and their IRIs are not visible. This means that it is straightforward to design project-specific Knora ontologies that are compatible with standard ontologies when represented in the simple schema.

KnarQL queries can use the simple schema; this greatly simplifies the queries themselves, without sacrificing their expressive power. Moreover, it makes it possible to use standard ontology entities in queries. For example, if a project-specific ontology defines a property `example:name` as a subproperty of `foaf:name`, a KnarQL query can use `foaf:name`, and treat the property's object as a literal.

3.2 KnarQL Syntax and Semantics

Syntactically, a KnarQL query is a SPARQL `CONSTRUCT` query. Thus it supports arbitrarily complex graph patterns. One could, for example, search for persons whose works have been published by a publisher that is located in a particular city. A `CONSTRUCT` query also allows the client to specify, for each resource that matches the search criteria, which values of the resource should be returned in the search results. The syntax of a KnarQL `CONSTRUCT` query

has some restrictions, which are aimed mainly at simplifying the implementation:

- Named graphs, blank nodes, and `LIMIT` may not be used.
- The `WHERE` clause may contain only triple patterns, `FILTER`, and `UNION`. For example, `BIND` is not allowed.
- A restricted set of `FILTER` expressions and functions is supported.
- A `UNION` or `OPTIONAL` may not contain a nested `UNION` or `OPTIONAL`.

KnarQL also imposes some requirements on the semantics of queries:

- The `CONSTRUCT` clause must designate one variable that represents the *main resource* in each search result. This is done using the predicate `knora-api:isMainResource`.
- The Knora API server must be able to determine the type of each variable used in the query. In the current implementation, the `WHERE` clause must specify variable types explicitly by adding statements that are, in effect, type annotations. In a future version, type inference may make these annotations unnecessary.

While the behaviour of `ORDER BY` and `OFFSET` is undefined in a SPARQL `CONSTRUCT` query (which by definition returns an unordered set of triples), KnarQL uses them for the paging of query results. The client can use `ORDER BY` with one or more variables to determine the order in which results will be returned, and `OFFSET` to specify which page of results should be returned. The number of results per page is configurable in the Knora API server, and cannot be controlled by a KnarQL query. The first page is `OFFSET 0`, the next page is `OFFSET 1`, and so on.

3.3 KnarQL Response Format

The response format is JSON-LD, which is our preferred format for information exchange in the Knora API. Unlike the response to a SPARQL CONSTRUCT request, which is simply a set of triples, the response to a KnarQL query provides data in a hierarchical structure, containing one or more resources with properties, which can point to and contain other resources with their properties. The API thus translates the flat structure of a set of triples into something that, in our experience, is more useful in web application development, while remaining generic. Moreover, the Knora API reuses this

same generic response format for all KnarQL responses as well as for other API responses.

Listing 1: Knora API response to a resource request

```
{
  "@type" : "schema:ItemList",
  "schema:itemListElement" : [ {
    "@id" : "http://rdfh.ch/8a0b1e75",
    "@type" : "incunabula:page",
    "incunabula:partOf" : {
      "@id" : "http://rdfh.ch/c5058f3a"
    },
    "incunabula:seqnum" : 1,
    "rdfs:label" : "a1r, Titelblatt"
  }, {
    "@id" : "http://rdfh.ch/4f11adaf",
    "@type" : "incunabula:page",
    "incunabula:partOf" : {
      "@id" : "http://rdfh.ch/c5058f3a"
    },
    "incunabula:seqnum" : 2,
    "rdfs:label" : "a1v, Titelblatt, Rückseite"
  } ],
  "schema:numberOfItems" : 2,
  "@context" : {
    "rdf" : "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "knora-api" : "http://api.knora.org/ontology/knora-api/simple/v2#",
    "schema" : "http://schema.org/",
    "rdfs" : "http://www.w3.org/2000/01/rdf-schema#",
    "incunabula" : "http://0.0.0.0:3333/ontology/0803/incunabula/simple/v2#"
  }
}
```

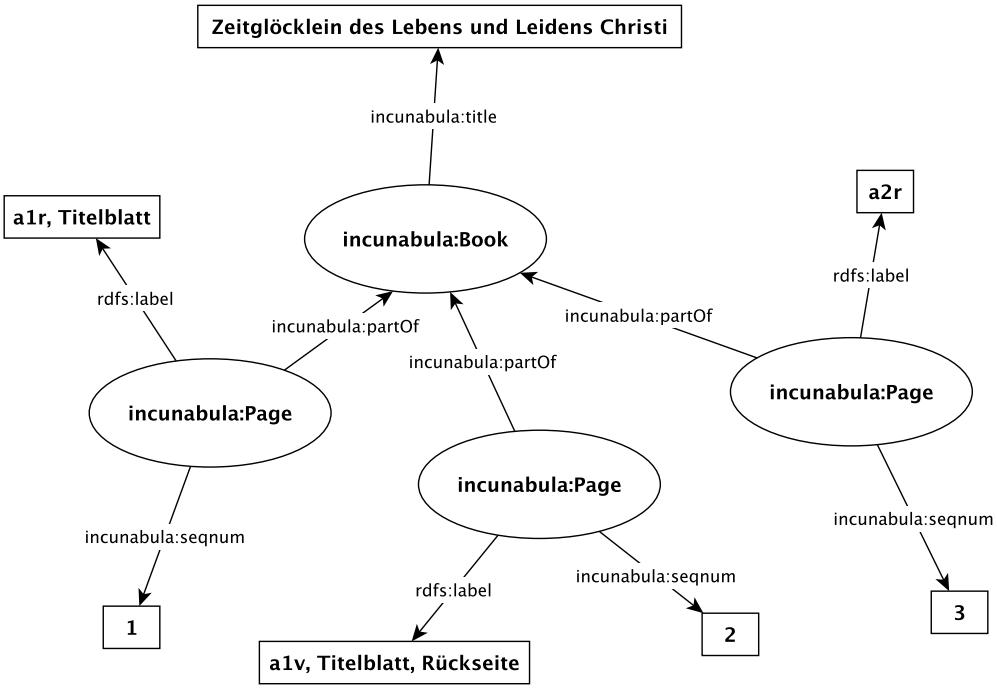
Listing 1 shows the representation of two resources. As an example, we chose two resources of type `page` defined in an ontology called `incunabula`.[1] These represent the first two pages of the book *Zeitglöcklein des Lebens und Leidens Christi* (Figure 1). Each page resource refers to the IRI of the book that it is part of.

4 Processing and Execution of a KnarQL Query

The Knora API server processes each KnarQL query, converting it to to SPARQL that is actually executed by the triplestore. The generated SPARQL is more complex than the provided KnarQL query, and deals with Knora's internal data structure. A KnarQL query is thus a virtual query. In our current implementation, each KnarQL query is converted to two SPARQL queries to improve performance. First, the a 'prequery' is generated from the given KnarQL, to identify a page of matching resources. Then a 'main query' is executed, to retrieve the requested values of those resources.

[1] The Incunabula project contains 19 early printed books from the late 15th century.

Figure 1: Early Printed Books from the Incunabula Project



4.1 Prequery

The prequery is a SPARQL SELECT query that uses `OFFSET` and `LIMIT` to enforce paging. Its main purpose is to obtain the IRIs of a particular page of resources that match the given search criteria. To generate the prequery, the KnarQL query's `WHERE` clause is adapted to the greater complexity of the internal ontology schema. To enforce the paging of results, the prequery's `LIMIT` is set to a value defined in the Knora API server's configuration (by default 25), and its `OFFSET` is set according to the results page number requested. The prequery's results consist of an ordered list of IRIs referring to matching resources and values as well as to dependent resources (links from matching resources to other resources). To ensure that paging functions correctly, the prequery always returns results in a deterministic order.

4.2 Main Query

The IRIs returned by the prequery are then used in the main query. The main query is a SPARQL CONSTRUCT query that obtains more detailed information about matching resources and values. It does not need to consider

the search criteria, since this has already been taken care of. The results of the main query are filtered according to the client's permissions. A resource appears in the results only if the client has the permissions to see all values and dependent resources specified in the query. If this is not the case, an empty resource will be shown instead, informing the client that it has insufficient permissions to see the actual matching resource. Thus if a full page of results was found, a full page is returned, perhaps with some empty resources. The client can then request another page of results. If a full page is not returned, it means that this is the last page.

5 Use Case from the Bernoulli-Euler Online Project

One project that is using KnarQL is Bernoulli-Euler Online (BEOL), a digital edition project focusing on 17^{th}- and 18^{th}-century primary sources in mathematics. BEOL integrates written sources relating to members of the Bernoulli dynasty and Leonhard Euler into one platform based on Knora. The data is stored in an RDF triplestore managed by the Knora API server. The BEOL web site provides a user interface that enables users to search and view these texts in a variety of ways. It offers a menu of common queries that internally generate KnarQL using templates, and the user can also build a custom query using a graphical search interface, which also generates KnarQL internally.

5.1 Example 1

Most of the texts that are currently integrated in the BEOL platform are letters exchanged between mathematicians. On the project's landing page, we would like to present the letters arranged by their authors and recipients. With KnarQL, we do not need to make a custom API route for this kind of query in Knora. We can simply write a KnarQL template that contains the correspondents as variables.

As shown in Listing 2, the IRIs identifying the correspondents do not have to be submitted with the KnarQL query. In order to get these IRIs, additional queries would have to be executed first. Instead, their IAF[2] identifiers can be used. Each person is represented as a resource that has a property

[2]A person can be uniquely identified by means of an Integrated Authority File (IAF) id. An IAF identifier can be resolved with the webservice of the German National library: `http://d-nb.info/gnd/118531379`.

`beol:hasIAFIdentifier` with a unique value. In the template we can simply substitute the placeholders `iaf1` and `iaf2` with actual IAF identifiers (see lines 21 and 28 in Listing 2).[3] For instance, we can choose the identifiers for Leonhard Euler (118531379) and Christian Goldbach (118696149).

Given the two IAF identifiers, the KnarQL query gets all letters exchanged between these two persons, sorted by date. By 'exchanged', we mean that either individual can be the author or recipient of a letter. This can be achieved by using a SPARQL variable representing a property that connects letters and persons, and by restricting it to the properties `beol:hasAuthor` and `beol:hasRecipient` by means of a `FILTER`. This excludes, for example, the property `beol:mentionsPerson`, which indicates that a certain person is mentioned in a letter.

Listing 2: KnarQL query template

```
1  PREFIX beol: <http://beol.dasch.swiss/ontology/0801/beol/simple/v2#>
2  PREFIX knora-api: <http://beol.dasch.swiss/ontology/knora-api/simple/v2#>
3
4  CONSTRUCT {
5      ?letter knora-api:isMainResource true .
6
7      ?letter beol:creationDate ?date .
8
9      ?letter ?linkingProp1  ?person1 .
10
11     ?letter ?linkingProp2  ?person2 .
12
13 } WHERE {
14     ?letter a beol:letter .
15
16     ?letter ?linkingProp1  ?person1 .
17     FILTER(?linkingProp1 = beol:hasAuthor || ?linkingProp1 = beol:hasRecipient )
18
19     ?person1 a beol:person .
20
21     ?person1 beol:hasIAFIdentifier "${iaf}" .
22
23     ?letter ?linkingProp2 ?person2 .
24     FILTER(?linkingProp2 = beol:hasAuthor || ?linkingProp2 = beol:hasRecipient )
25
26     ?person2 a beol:person .
27
28     ?person2 beol:hasIAFIdentifier "${iaf}" .
29
30     ?letter beol:creationDate ?date .
31
32 } ORDER BY ?date
33 OFFSET ${offset}
```

The BEOL project's landing page combines the IAF identifiers of the main correspondents and generates KnarQL queries using the KnarQL template. By changing the value of the offset (starting with 0), all the letters belonging

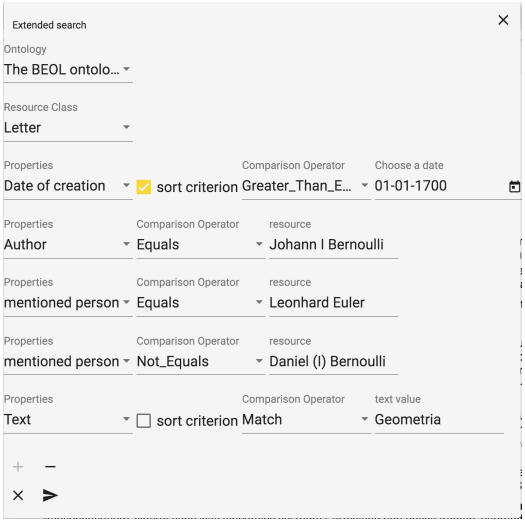
[3] KnarQL type annotations are omitted in this example for brevity.

to a correspondence can be fetched page by page and integrated into one list of results.

5.2 Example 2

Users create can also create custom queries that are not based on a predefined template. For this purpose, we developed a GUI widget that generates KnarQL, but does not require the user to write any code.

Figure 2: Advanced Search Widget



In Figure 2, the user created a query to search for all letters that were written after January 1st 1700 by Johann I Bernoulli, and that mention Leonhard Euler but not Daniel I Bernoulli, and contain the word 'Geometria' (exact word match). The results are ordered by date. The GUI widget generates a KnarQL query based on the search criteria (Listing 3).

Listing 3: KnarQL query generated from GUI widget

```
PREFIX beol: <http://beol.dasch.swiss/ontology/0801/beol/simple/v2#>
PREFIX knora-api: <http://beol.dasch.swiss/ontology/knora-api/simple/v2#>

CONSTRUCT {
    ?letter knora-api:isMainResource true .

    ?letter beol:creationDate ?date .
    ?letter beol:hasAuthor <http://rdfh.ch/biblio/Johann_I_Bernoulli> .
    ?letter beol:mentionsPerson <http://rdfh.ch/biblio/Leonhard_Euler> .
} WHERE {
    ?letter a beol:letter .

    ?letter beol:creationDate ?date .

```

```
15      FILTER(?date >= "GREGORIAN:1700-1-1")
16
17      ?letter beol:hasAuthor <http://rdfh.ch/biblio/Johann_I_Bernoulli> .
18
19      ?letter beol:mentionsPerson <http://rdfh.ch/biblio/Leonhard_Euler> .
20
21      FILTER NOT EXISTS {
22          ?letter beol:mentionsPerson <http://rdfh.ch/biblio/Daniel_I_Bernoulli> .
23      }
24
25      ?letter <beol:hasText> ?text .
26
27      FILTER knora-api:match(?text, "Geometria")
28  }
29  ORDER BY ?date
30  OFFSET 0
```

6 Conclusion

We have described a way for RDF-based humanities data repositories to combine powerful SPARQL-like search capabilities with the safety, convenience, scalability, and efficiency of an HTTP-based API. Our approach is to implement a virtual query language, based on SPARQL but processed by an API server rather than sent directly to a triplestore. By introducing this layer of abstraction between the client and the triplestore, the API server gains control over the actual SPARQL code that the triplestore runs, as well as over the data that is returned to the client. It can therefore enforce permissions, take into account the versioning of data, and ensure that queries use a scalable, efficient SPARQL query design. Moreover, this approach allows data to be stored in a form that is suitable for long-term preservation, but served in a form that is suitable for web application development. Thus it contributes to two common goals in digital humanities: making data accessible and interoperable while also ensuring its longevity.

References

[1] Lukas Rosenthaler, Peter Fornaro, and Claire Clivaz. 'DASCH: Data and Service Center for the Humanities'. In: *Digital Scholarship in the Humanities* 30 (2015), pp. i43–i49. DOI: 10.1093/llc/fqv051.