Semantic Web 1 (0) 1–5
IOS Press

Gravsearch: transpiling SPARQL to query humanities data

Tobias Schweizer [a,b] and Benjamin Geer [a,c]
[a] *Data and Service Center for the Humanities, Universität Basel, Bernoullistrasse 32, 4056 Basel, Switzerland*
[b] *E-mail: t.schweizer@unibas.ch*
[c] *E-mail: benjaminlewis.geer@unibas.ch*

Editors: First Editor, University or Company name, Country; Second Editor, University or Company name, Country
Solicited reviews: First Solicited Reviewer, University or Company name, Country; Second Solicited Reviewer, University or Company name, Country
Open reviews: First Open Reviewer, University or Company name, Country; Second Open Reviewer, University or Company name, Country

Abstract. It has become common for humanities data to be stored as RDF, but available technologies for querying RDF data, such as SPARQL endpoints, have drawbacks that make them unsuitable for many applications. Gravsearch (Virtual Graph Search), a SPARQL transpiler developed as part of a web-based API, is designed to support complex searches that are desirable in humanities research, while avoiding these disadvantages. It does this by introducing server software that mediates between the client and the triplestore, transforming an input SPARQL query into one or more queries executed by the triplestore. This design suggests a practical way to go beyond some limitations of the ways that RDF data has generally been made available.

Keywords: SPARQL, humanities, querying, qualitative data, API

1. Introduction

Gravsearch is a SPARQL transpiler facilitating the query of humanities data stored as RDF. The storage and publication of humanities data as RDF has become common, but there is a lack of appropriate technologies for searching this data for items and relationships that are of interest to humanities researchers. SPARQL endpoints are one option, but they present a number of drawbacks. Dealing with humanities-focused data structures in SPARQL can be cumbersome, and there is no support for permissions or for versioning of data. Queries that may return huge results also pose scalability problems. A technical solution to these problems is proposed here. Gravsearch aims to provide the power and flexibility of a SPARQL endpoint, while providing better support for humanities data, and integrating well into a developer-friendly web-based API. Its basic design is of broad relevance, because it suggests a practical way to go beyond some limitations of the ways that humanities data has generally been made searchable.

While a SPARQL endpoint accepts queries that are processed directly by an RDF triplestore, a Gravsearch query is a virtual SPARQL query, i.e. it is processed by a server application, which translates it into one or more SPARQL queries to be processed by the triplestore. Therefore, it can offer better support for humanities-focused data structures, such as text markup and calendar-independent historical dates. More generally, a Gravsearch query can use data structures that are simpler than the ones used in the triplestore, thus improving ease of use. A virtual query also allows the application to filter results according to user permissions, enforce the paging of results to improve scalability, take into account the versioning of data in the triplestore, and return responses in a form that is more convenient for web application development. The input SPARQL is independent of the triplestore implementation used, and the

transpiler backend generates vendor-specific SPARQL as needed, taking into account the triplestore's implementation of inference, full-text searches, and the like. Instead of simply returning a set of triples, a Gravsearch query can produce a JSON-LD response whose structure facilitates web application development.

Gravsearch has been developed as part of Knora (Knowledge Organization, Representation, and Annotation), an application developed by the Data and Service Center for the Humanities (DaSCH) [1] to ensure 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. Knora provides a web-based API that allows data to be queried and updated, and supports the creation of virtual research environments that can work with heterogeneous research data from different disciplines.[1]

1.1. Problem definition

Knora required a search language that distinguishes between data as it is stored and data as it is returned to client applications. For example, in humanities research, it is useful to search for dates independently of the calendar in which they are written in source materials. Knora makes this possible by storing dates as Julian Day Numbers (JDNs), a calendar-independent astronomical representation. Clients should be able to submit search requests for, say, texts published within a date range in the Gregorian calendar. Knora should then find texts with dates within the corresponding JDN range, and return results showing those dates in the relevant calendars. The client should not have to send or receive JDNs itself, or to submit the complex SPARQL that must be generated to search for these date ranges in the triplestore.

Another example concerns text with markup, which Knora stores as RDF data. A humanities researcher might wish to search a large number of texts for, say, paragraph tags that contain a particular word. Knora could optimise this search by using a full-text search index. This also involves rather complex SPARQL, which is partly specific to the type of indexing software being used. The client should not have to deal with these details.

[1]For more information on Knora, see https://www.knora.org. A list of projects implemented, planned, or under development using Knora can be found at https://dasch.swiss/projects/.

SPARQL lacks other features required in this context. Knora must restrict access to data according to user permissions. It also implements a system for versioning data in the triplestore, such that the most recent version is returned by default, but the version history of resources can be requested. To improve scalability, Knora should enforce the paging of search results, rather than leaving this up to the client as in SPARQL.

1.2. Related work

One way of making RDF data publicly available and queryable is by means of a SPARQL endpoint. Two prominent examples are DBpedia [2] and Europeana [3]. While SPARQL endpoints offer great flexibility and allow for complex queries, their drawbacks have been criticised. In a widely cited blog post [4], Dave Rogers argues that SPARQL endpoints are an inherently poor design that cannot possibly scale, and that RESTful APIs should be used instead. For example, a SPARQL endpoint allows a client to request all the data in the repository; this could easily place unreasonable demands on the server, particularly if many such requests are submitted concurrently.

GraphQL [5] 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 [6], 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 Rogers's blog post [7]. 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 limitations that make them unsuitable for Knora and for humanities data in general. They assume that the data structures in the triplestore are the same as the ones to be returned to the client. They are also 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. They do not enforce the paging of results, but leave this to the client. And 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). These requirements led us to develop a different approach.

2. A hybrid between a SPARQL endpoint and a web API

One option would be to create a domain-specific language, but it was simpler to use SPARQL, leveraging its standardisation and library support, while integrating it into Knora's web API. Gravsearch therefore accepts as input a subset of standard SPARQL query syntax, and it requires queries to follow certain additional rules.

Gravsearch is thus a hybrid between a SPARQL endpoint and a web API, aimed at combining the advantages of both. To enable query results to be processed in a generic way by the application, Gravsearch accepts only SPARQL CONSTRUCT queries, which return sets of triples. The Knora API server processes the input query, translating it into one or more SPARQL queries that are executed by the triplestore, then processes and transforms the triples returned by the triplestore to construct the response. This extra layer of processing enables Gravsearch to avoid the disadvantages of SPARQL endpoints and to provide additional humanities-focused features. Results are filtered according to the user's permissions, the versioning of data in the triplestore is taken into account (only the most recent version of the data is returned), and scalability is improved by returning results in pages of limited size.

Although SELECT queries are not supported, if tabular output is desired, (e.g. for statistical analysis), the results of a CONSTRUCT query can be converted into a table by combining results pages and converting the RDF output to tabular form. This could be done either in the client or on the server.

The source code of the Gravsearch implementation is available on GitHub[2], and the design documentation can be consulted online[3].

2.1. Ontology schemas

A design goal of Gravsearch 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. The term *internal schema* refers to the structures that are actually in the triplestore, and *external schema* refers to a view that transforms these structures in some way for use in a web API.

In the internal schema, the smallest unit of research data is a Knora `Value`, which is an RDF entity that has an IRI. If a client wishes to update a value via the Knora API, it needs to know the value's IRI. However, a Knora value also contains information that is represented in a way that is not convenient for clients to manipulate (e.g. dates are stored as JDNs, as mentioned above). Therefore, Knora provides an external schema called the *complex schema*, in which each value has an IRI, but its contents are represented in more convenient form (e.g. calendar dates are used instead of JDNs).

For clients that need a read-only view of the data, Knora provides a *simple external schema*, in which Knora values are represented as literal datatypes such as `xsd:string`, and the metadata attached to each value is hidden. An advantage of the simple schema is that it facilitates the use of standard ontologies such as Dublin Core,[4] in which values are represented as datatypes without their own metadata. For example, if a property is defined in Knora as a subproperty of `dcterms:title`, its object in Knora will internally be a Knora `TextValue` with attached metadata, but a Gravsearch query in the simple schema can treat it as a literal, in keeping with its definition in `dcterms`.

Additional external schemas could be added in future. Only the internal schema is used in the triplestore; Knora converts data and ontology entities between internal and external schemas during request processing. Gravsearch queries can thus be written in either of Knora's external schemas, and results can also be returned in either of these schemas.

2.2. Permissions

In Knora, each resource and each value has user and group permissions attached to it. Internally, permissions are represented as string literals in a compact format that optimises query performance. For example, a Knora value could contain this triple:

[2] https://github.com/dhlab-basel/Knora
[3] https://docs.knora.org
[4] https://www.dublincore.org/schemas/rdfs/

```
<http://rdfh.ch/0001/R5qJ6oPZPV>
  knora-base:hasPermissions
  "V http://rdfh.ch/groups/00FF/reviewer" .
```

This means that the value can be viewed by members of the specified group. With a SPARQL endpoint, there would be no way to prevent other users from querying the value. Therefore, the application must filter query results according to user permissions.

To determine whether a particular user can view the value, Knora must compute the intersection of the set of groups that the user belongs to and the set of groups that have view permission on the value. If not, Knora removes the value (and the resource that contains it) from the results of the Gravsearch query.

2.3. Versioning

Internally, a resource is connected only to the current version of each of its values. Each value version is connected to the previous version via the property `previousVersion`, so that the versions form a linked list. When a client requests a single resource with its values via the Knora API, the client can specify a version timestamp. Knora then generates a SPARQL query that traverses the linked list to retrieve the values that the resource had at the specified time.

In ordinary use, Gravsearch should query only current data. This is easily achieved, because the only way to obtain a value in Gravsearch is to follow the connection between the resource and the value, which is always the current version. Knora's external ontology schemas do not expose the version history data structure at all (e.g. they do not provide the property `previousVersion`). Therefore, the client cannot accidentally query a past version of a value, which would be possible with a SPARQL endpoint.

As a future development, it may also be feasible to provide a timestamp with a Gravsearch query, to obtain results that existed at the specified time.

2.4. Gravsearch syntax and semantics

Syntactically, a Gravsearch query is a SPARQL `CONSTRUCT` query. Thus it supports arbitrarily complex search criteria. 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.

Results are returned by default as a JSON-LD array, with one element per search result. Each search result contains the 'main' or top-level resource that matched the query. If the query requests other resources that are connected to the main resource, these are nested as JSON-LD objects within the main resource. To make this possible, a Gravsearch query must specify (in the `CONSTRUCT` clause) which variable refers to the main resource. The resulting tree structure is generally more useful to web application clients than the flat set of RDF statements returned by SPARQL endpoints.

Gravsearch uses the SPARQL constructs `ORDER BY` and `OFFSET` to enable the client to step through pages of search results. (These constructs are not meaningful in a standard SPARQL `CONSTRUCT` query, which by definition returns an unordered set of triples.) 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 application's settings, and cannot be controlled by the client.

2.5. Processing and execution of a Gravsearch query

Knora processes each Gravsearch query, converting it to one or more SPARQL queries that are actually executed by the triplestore. The generated SPARQL is considerably more complex than the provided Gravsearch query, and deals with data structures in the internal schema. In our current implementation, each Gravsearch query is converted to two SPARQL queries to improve performance. First, a `SELECT` query is generated, to identify a page of matching resources. Then a `CONSTRUCT` query is generated, to retrieve the requested values of those resources. The API server is free to use a SPARQL design that best suits the performance characteristics of the triplestore; clients and users need not know how to do this.

2.5.1. Type checking and inference

SPARQL does not provide type checking; if a SPARQL query uses a property with an object that is not compatible with the property definition, the query will simply return no results.

However, Gravsearch needs know the types of the entities used in a query so it can generate the correct SPARQL. Specifically, it needs to know type of the subject and object of each statement, and the type that is expected as the object of each predicate.

To get this information, Gravsearch implements type inference, based on the types that are actually used in the query, as well as information from ontologies in the triplestore. Knora requires each research project to provide one or more ontologies defining the OWL classes and properties used in its data. These ontologies must be derived from the Knora base ontology, which provides common data structures such as the low-level value types that Knora supports. Gravsearch has access to these ontologies, and uses them to infer the types of entities designated by variables and IRIs in the input query.

Appendix A shows an input query in the simple schema. It searches for books that have a particular publisher (identified by IRI), and returns them along with the family names of all the persons that have some connection with those books (e.g. as author or editor).

In this example, the definition of the property `hasPublisher` specifies that its object must be a `Publisher`, allowing Gravsearch to infer the type of the resource identified by the specified IRI. Similarly, the definition of `hasFamilyName` specifies that its subject must be a `Person` and its object must be a Knora `TextValue`; this allows the types of `?person` and `?familyName` to be inferred. Once the type of `?person` is known, the object type of `?linkProp` can then be inferred.

Unlike a SPARQL endpoint, if Gravsearch cannot determine the type of an entity, or finds that an entity has been used inconsistently (i.e. with two different types), it returns an error message rather than an empty response.

3. Use case from the Bernoulli-Euler Online project

One project that is using Gravsearch is Bernoulli-Euler Online (BEOL),[5] a digital edition project focusing on 17th- and 18th-century primary sources in mathematics. BEOL integrates written sources relating to members of the Bernoulli dynasty and Leonhard Euler into a web application based on Knora, with data stored in an RDF triplestore. 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 Gravsearch using templates, and the user can also build a custom query using a graphical search interface, which also generates Gravsearch internally.

[5]https://beol.dasch.swiss/.

3.1. Example 1: finding correspondence between two mathematicians

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 Gravsearch, it is not necessary to make a custom API operation for this kind of query in Knora. Instead, a Gravsearch template can be used, with variables for the correspondents.

Appendix B shows a template for a Gravsearch query that finds all the letters exchanged between two persons. Each person is represented as a resource in the triplestore. It would be possible to use the IRIs of these resources to identify mathematicians, but since these IRIs are not yet stable during development, it is more convenient to use the property `beol:hasIAFIdentifier`, whose value is an Integrated Authority File (IAF) identifier (maintained by the German National Library), a number that uniquely identifies that person. This example thus illustrates searching for resources that have links to other resources that have certain properties. The user chooses the names of two mathematicians from a menu in a web browser, and the user interface then processes the template, substituting the IAF identifiers of those two mathematicians for the placeholders `${iaf1}` and `${iaf2}`. The result of processing the template is a Gravsearch query, which the user interface submits to the Knora API server. This query specifies that the author and recipient of each matching letter must have one of those two IAF identifiers. The results are sorted by date. The page number ${offset} is initially set to `0`; as the user scrolls, the page number is incremented and the query is run again to load more results.

This query is simple enough to be written in the simple schema. For example, this allows the object of `beol:hasIAFIdentifier` to be treated as a string literal. Internally, this is an object property. Its object is an entity belonging to the class `knora-base:TextValue`, and has predicates and objects of its own. This extra level of complexity is hidden from the client in the simple schema. After we substitute the IAF identifiers of Leonhard Euler and Christian Goldbach for the placeholders in the template, the input query contains:

```
?author beol:hasIAFIdentifier
  ?authorIAF .
FILTER(?authorIAF =
  "(DE-588)118531379" ||
  ?authorIAF = "(DE-588)118696149")
```

Gravsearch transforms these two lines to the following SPARQL:

```
?author beol:hasIAFIdentifier ?authorIAF .
?authorIAF knora-base:isDeleted false .
?authorIAF knora-base:valueHasString
  ?authorIAF__valueHasString .
FILTER(?authorIAF__valueHasString =
  "(DE-588)118531379"^^xsd:string ||
  ?authorIAF__valueHasString =
  "(DE-588)118696149"^^xsd:string)
```

Since values in Knora can be marked as deleted, the generated query uses `knora-base:isDeleted false` to exclude deleted values. It then uses the generated variable `?authorIAF__valueHasString` to match the content of the `TextValue`.

3.2. Example 2: a user interface for creating queries

Users can also create custom queries that are not based on a predefined template. For this purpose, a user-interface widget generates Gravsearch, without requiring the user to write any code (Appendix D).

For example, a user can create a query to search for all letters written since 1 January 1700 CE (the user specifies the Gregorian calendar) by Johann I Bernoulli, that mention Leonhard Euler but not Daniel I Bernoulli, and that contain the word *Geometria*. The user can choose to order the results by date. The web-based user interface generates a Gravsearch query based on the search criteria (Appendix C).

In generating SPARQL to perform the requested search, the Knora API server converts the date comparison to one that uses a JDN. In Knora, every date is stored as a date range with a particular precision (year, month, or day), whose start and end points are JDNs. In the example, the input SPARQL requests a date greater than a date literal in the Gregorian calendar:

```
FILTER(?date >=
  "GREGORIAN:1700-1-1"^^knora-api:Date)
```

The Gravsearch transpiler converts this to a JDN comparison. The Gregorian date 1 January 1700 is converted to the JDN 2341973. In the generated SPARQL, a matching date's end point must be greater than or equal to that JDN:

```
?date knora-base:valueHasEndJDN
  ?date__valueHasEndJDN .
FILTER(?date__valueHasEndJDN >=
  "2341973"^^xsd:integer)
```

To specify that the text of the letter must contain the word *Geometria*, the input SPARQL uses the function `knora-api:match`, which is provided by Gravsearch:

```
FILTER knora-api:match(?text,
  "Geometria")
```

The transpiler converts this function to triplestore-specific SPARQL that performs the query using a full-text search index. For example, with the GraphDB triplestore using the Lucene full-text indexer, the generated query contains:

```
?text knora-base:valueHasString
  ?text__valueHasString .
?text__valueHasString
  lucene:fullTextSearchIndex
  "Geometria"^^xsd:string .
```

3.3. Example 3: Searching for text markup

Here were are looking for a letter containing the word *Richtigkeit* in text that is marked up as a paragraph.

Knora stores text markup as 'standoff markup': each markup tag is represented as an entity in the triplestore, with start and end positions referring to a substring in the text. This makes it possible for queries to combine criteria referring to text markup with criteria referring to other entities in the triplestore, including links within text markup that point to RDF resources outside the text.

To search for text markup, the input query must be written in the complex schema. The input query uses the `matchInStandoff` function provided by Gravsearch:

```
?text knora-api:valueAsString
  ?textStr .
?text knora-api:textValueHasStandoff
  ?standoffParagraphTag .
?standoffParagraphTag a
  standoff:StandoffParagraphTag .
FILTER knora-api:matchInStandoff(
  ?textStr,
  ?standoffParagraphTag,
  "Richtigkeit")
```

Gravsearch translates this `FILTER` into two operations:

1. An optimisation that searches in the full-text search index to find all texts containing this word.
2. A regular expression match that determines whether, in each text, the word is located within a substring that is marked up as a paragraph.

The resulting generated SPARQL looks like this[6]:

```
?textStr lucene:fullTextSearchIndex
  "Richtigkeit"^^xsd:string .
?standoffTag
  knora-base:standoffTagHasStart
  ?standoffTag__start .
?standoffTag
  knora-base:standoffTagHasEnd
  ?standoffTag__end .
BIND(substr(?textStr,
  ?standoffTag__start + 1,
  ?standoffTag__end -
    ?standoffTag__start)
  AS ?standoffTag__markedUp)
FILTER(regex(?standoffTag__markedUp,
  "Richtigkeit", "i"))
```

4. Conclusion

This article has described a way for RDF-based humanities data repositories to provide powerful SPARQL search capabilities with additional humanities-focused features, and with the safety, convenience, scalability, and efficiency of a web API. This approach involves a virtual SPARQL query that is processed by an API server rather than sent directly to a triplestore. This allows the API server to support data structures that are relevant to humanities research, as well as to enforce permissions, require search results to be paged, take into account the versioning of data, and better optimise the underlying SPARQL queries. Moreover, this approach allows us to store data in a form that is suitable for long-term preservation, while serving it 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.

[6]Knora uses 0-based indexes in standoff markup, but SPARQL uses 1-based indexes: https://www.w3.org/TR/xpath-functions/#func-substring.

Acknowledgements

This work was supported by the Swiss National Science Foundation (166072) and the Swiss Data and Service Center for the Humanities.

Appendix A. A simple Gravsearch query

```
PREFIX example: <http://example.org/ontology/0001/example/simple/v2#>
PREFIX knora-api: <http://api.knora.org/ontology/knora-api/simple/v2#>

CONSTRUCT {
   ?book knora-api:isMainResource true .
   ?book ?linkProp ?person .
   ?person example:hasFamilyName ?familyName .
} WHERE {
   ?book a example:Book ;
     example:hasPublisher <http://rdfh.ch/0001/B3lQa6tSymIq7> ;
     ?linkProp ?person .
   ?person example:hasFamilyName ?familyName .
}
OFFSET 0
```

Listing 1: A Gravsearch template

Appendix B. A Gravsearch template

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

CONSTRUCT {
  ?letter knora-api:isMainResource true .
  ?letter beol:creationDate ?date .
  ?letter beol:hasAuthor ?author .
  ?letter beol:hasRecipient ?recipient .
} WHERE {
  ?letter a beol:letter .
  ?letter beol:creationDate ?date .

  ?letter beol:hasAuthor ?author .
  ?author beol:hasIAFIdentifier ?authorIAF .
  FILTER(?authorIAF = "${iaf1}" || ?authorIAF = "${iaf2}")

  ?letter beol:hasRecipient ?recipient .
  ?recipient beol:hasIAFIdentifier ?recipientIAF .
  FILTER(?recipientIAF = "${iaf1}" || ?recipientIAF = "${iaf2}")
}
ORDER BY ?date
OFFSET ${offset}
```

Listing 2: A Gravsearch template

Appendix C. A user-generated query

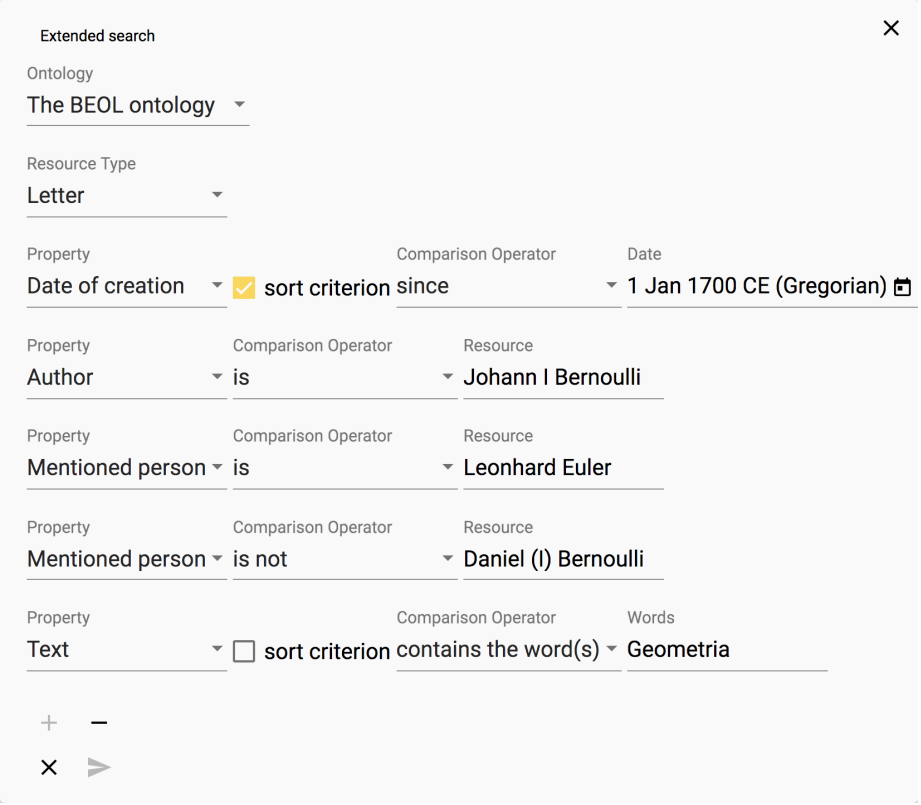
```
PREFIX beol: <http://beol.dasch.swiss/ontology/0801/beol/simple/v2#>
PREFIX knora-api: <http://api.knora.org/ontology/knora-api/simple/v2#>

CONSTRUCT {
  ?letter knora-api:isMainResource true .
  ?letter beol:creationDate ?date .
} WHERE {
```

```
  ?letter a beol:letter .

  ?letter beol:creationDate ?date .
  FILTER(?date >= "GREGORIAN:1700-1-1"^^knora-api:Date)

  ?letter beol:hasAuthor <http://rdfh.ch/biblio/Johann_I_Bernoulli> .
  ?letter beol:mentionsPerson <http://rdfh.ch/biblio/Leonhard_Euler> .

  FILTER NOT EXISTS {
    ?letter beol:mentionsPerson <http://rdfh.ch/biblio/Daniel_I_Bernoulli> .
  }

  ?letter beol:hasText ?text .
  FILTER knora-api:match(?text, "Geometria")
}
ORDER BY ?date
OFFSET ${offset}
```

Listing 3: A user-generated query

Appendix D. GUI widget

Figure 1. Advanced Search Widget



Appendix E. Searching for text markup

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

CONSTRUCT {
   ?letter knora-api:isMainResource true .
   ?letter beol:hasText ?text .
} WHERE {
   ?letter a beol:letter .
   ?letter beol:hasText ?text .
   ?text knora-api:valueAsString ?textStr .
   ?text knora-api:textValueHasStandoff ?standoffParagraphTag .
   ?standoffParagraphTag a standoff:StandoffParagraphTag .
   FILTER knora-api:matchInStandoff(?textStr, ?standoffParagraphTag, "Richtigkeit")
}
```

Listing 4: A user-generated query

References

[1] L. Rosenthaler, P. Fornaro and C. Clivaz, DASCH: Data and Service Center for the Humanities, *Digital Scholarship in the Humanities* **30** (2015), i43–i49. doi:10.1093/llc/fqv051.

[2] DBpedia, Virtuoso SPARQL Query Editor. http://dbpedia.org/sparql.

[3] Europeana, Virtuoso SPARQL Query Editor. http://sparql.europeana.eu/.

[4] D. Rogers, The Enduring Myth of the SPARQL Endpoint. https://daverog.wordpress.com/2013/06/04/the-enduring-myth-of-the-sparql-endpoint/.

[5] GraphQL, GraphQL. https://www.howtographql.com/.

[6] HyperGraphQL, HyperGraphQL. http://hypergraphql.org/.

[7] S. Klarman, Querying DBpedia with GraphQL. https://medium.com/@sklarman/querying-linked-data-with-graphql-959e28aa8013.