Semantic Web 1 (0) 1–5
IOS Press

Gravsearch: transforming 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. RDF triplestores have become an appealing option for storing and publishing humanities data, but available technologies for querying this data have drawbacks that make them unsuitable for many applications. Gravsearch (Virtual Graph Search), a SPARQL transformer 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 transforms SPARQL queries and results to facilitate the use of humanities data stored as RDF.[1] Efforts have been made to use RDF triplestores for the storage and publication of humanities data [1, 2], but there is a lack of appropriate technologies for searching this data for items and relationships that are of interest to humanities researchers. A SPARQL endpoint backed directly by the triplestore is one option, but presents a number of drawbacks. It can be cumbersome in SPARQL to query certain data structures that are especially useful in the humanities, and there is no standard 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.

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 support for data structures that are especially useful in the humanities, such as text markup and calendar-independent historical dates, and are not included in RDF standards. 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 ver-

[1] The source code of the Gravsearch implementation, and its design documentation, are freely available online; see https://www.knora.org.

sioning 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 transformer 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.

The current implementation of Gravsearch is closely tied to a particular application, but its design is of more general interest to developers of RDF-based systems. It demonstrates that, for an application that manages data in an RDF triplestore and provides a web-based API, there is considerable value in a SPARQL-based API, which accepts SPARQL queries from the client, transforms them in the application, runs the transformed queries in the triplestore, and transforms the results before returning them to the client. It also suggests solutions to some of the practical issues that arise in the development of such a system. Consideration of this approach may lead others to develop similar application-specific tools, or to envisage more generic approaches to SPARQL transformation.[2]

1.1. Institutional and technical context

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) [5] to ensure the long-term availability and reusability of research data in the humanities.[3] The Swiss National Science Foundation (SNSF) requires researchers to have a plan for making their research data publicly accessible,[4] and the DaSCH was created to provide the necessary infrastructure to meet this requirement.

It not feasible or cost-effective for the DaSCH to maintain a multitude of different data formats and storage systems over the long term. Moreover, part of the DaSCH's mission is to make all the data it stores interoperable, so that it is possible to search for data across projects and academic disciplines in a generic way, regardless of the specific data structures used in each project. For example, many projects store text with markup, and it should be possible to search the markup of all these texts, regardless of whether they are letters, poems, books, or anything else. Moreover, when markup contains references to other data, it is useful for humanities researchers to search *through* the markup to the data it refers to, e.g. by searching for texts that refer to a person born after a certain date (as in the example in Section 2.2). It is therefore desirable to store text markup in the same way as other data, so they can be searched together.

The DaSCH must also mitigate the risks associated with technological and institutional change. The more its data storage is based on open standards, the easier it will be to migrate data to some other format if it becomes necessary to do so in future.

With these requirements in mind, the DaSCH has chosen to store all data in an RDF triplestore as far as possible (the main exceptions being images, audio, and video). RDF's flexibility allows it to accommodate all sorts of data structures. Its standardisation, along with the variety of triplestore implementations available, helps reduce the risks of long-term preservation. Even if RDF technology disappears over time, the RDF standards will make it possible to migrate RDF graphs to other types of graph storage.

Knora is therefore based on an RDF triplestore and a base ontology. The base ontology defines basic data types and abstract data structures; it is generic and does not make any assumptions about semantics.[5] Each research project using Knora must provide one or more ontologies defining the OWL classes and properties used in its data, and these ontologies must be derived from the Knora base ontology.

Knora is a server application written in Scala; it 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.[6]

The DaSCH's mission of long-term preservation requires it to minimise the risk of vendor lock-in. Knora must therefore, as far as possible, avoid relying on

[2] For some steps in this direction, see [3] and [4].

[3] The DaSCH as an institution is responsible for research data created by humanities projects. However, this does rule out the possibility that the technical solutions developed by the DaSCH are useful to other fields of research as well.

[4] http://www.snf.ch/en/theSNSF/research-policies/open_research_data/Pages/default.aspx

[5] For details of the Knora base ontology, see the documentation at https://www.knora.org.

[6] 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/.

any particular triplestore implementation, or on any type of data storage that has not been standardised and does not have multiple well-maintained implementations. This means that as long as there is no widely implemented RDF standard for some of the DaSCH's requirements, such as access control and versioning, Knora has to handle these tasks itself to ensure consistent behaviour, regardless of which triplestore is used.[7] It was therefore decided, for example, to use an in-band approach to storing permissions in a role-based access control model, and to store previous versions of values as linked lists, all in standard RDF.[8] Over time, if new RDF standards for these features are created and widely supported, it will become possible to remove them from Knora and allow the triplestore to handle them.

1.2. Problem definition

Knora's API required a query language that allows for complex searches in specific projects, as well as cross-project searches using ontologies shared by different projects, such as `FOAF`[9] or `dcterms`.[10] Researchers need to able be to find connections in their data across project boundaries, and developers need to present a project's data online in an interactive way to researchers and the interested public. There needs to be a flexible way to request particular graphs of data, not only for searches, but also to present combinations of resources that are known to be of interest to users. For example, the Bernoulli-Euler Online project (see Section 3) needed to present a manuscript page aligned with different layers of transcription, ranging from diplomatic transcriptions to translations [8]. To maximise the autonomy of web developers, Knora's API needs to make this possible without introducing project-specific API routes.

The query language also needs to allow queries to use data structures that are simpler than the ones stored in the triplestore. The DaSCH's requirement to have a generic storage system for humanities data, as outlined in Section 1.1, introduces complexities in the the RDF data structures used in the triplestore, and these complexities should be hidden from clients for the sake of usability.

For example, in humanities research, it is useful to search for dates independently of the calendar in which they are written in source materials. If an ancient Chinese manuscript records a solar eclipse or a sighting of a comet, and an ancient Greek manuscript records the same astronomical event at the same time, a search for texts mentioning the event and the date should find both texts, even though the date was written in different calendars. Astronomers and historians have long used a calendar-independent representation of dates, called Julian Day Numbers (JDNs), to facilitate such comparisons [9, 10]. A Julian Day Number is an integer, and can therefore be efficiently compared in a database query. Moreover, historical dates are often imprecise, e.g. when only the year is known, but not the month or day, or when a date is known only to fall within a certain range. Knora therefore stores each date in the triplestore as an instance of its `DateValue` class, which contains two Julian Day Numbers (representing the start and end of a date range), a precision (year, month, or day), and the calendar in which the date was originally recorded. This allows a SPARQL query to determine the chronological relationship between two dates, regardless of the calendar that they were originally created in.[11]

However, JDNs are clearly not a convenient representation for clients to use. Search requests and responses should use whatever calendars the client prefers. The client should not have to send or receive JDNs itself, or to deal with the complexities of comparing JDN ranges in SPARQL. In the Knora API, all dates are input and output as calendar dates; Knora converts between Julian Day Numbers and calendar dates using the International Components for Unicode library.[12]

[7] One exception is full-text search, which is implemented by the triplestore but is not standardised.

[8] For an overview of different approaches to RDF access control, see [6]. See The Fundamentals of Semantic Versioned Querying for a proposed versioning extension to RDF. The RDF*/SPARQL* reification proposal [7] may also lead to new ways of solving these problems.

[9] http://xmlns.com/foaf/spec/

[10] https://www.dublincore.org/specifications/dublin-core/dcmi-terms/

[11] Projects such as `GODOT` [11] approach this task differently, by creating reference entries for calendar dates identified by URIs that can then be used in online editions to allow for searches for a specific date. In our view, the approach of storing and comparing JDNs in the triplestore allows for more powerful searches, because JDNs can be compared using operators such as less than and greater than. In the future, we would also like to allow for the support of period terms. Projects such as `ChronOntology` [12] or `PeriodO` [13] relate period terms to timespans and geographical regions. Period terms can be referred to by means of URIs.

[12] http://site.icu-project.org/home

Another example concerns text with markup, which Knora stores as RDF data (see Section 3.3). A humanities researcher might wish to search a large number of texts for, say, a particular word marked up as a noun. 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.

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.

1.3. Related work

One way of making RDF data publicly available and queryable is by means of a SPARQL endpoint backed directly by a triplestore. Prominent examples include DBpedia [14], Wikidata [15], and Europeana [16]. While this approach offers great flexibility and allows for complex queries, its drawbacks have been criticised. In a widely cited blog post [17], 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 [18] 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 [19], 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 [20]. However, HyperGraphQL is designed to communicate directly with a SPARQL endpoint, and thus shares some of the limitations of SPARQL endpoints.

From our perspective, triplestore-backed 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 offer no standard 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. By supporting SPARQL syntax, it enables clients to submit queries based on complex graph patterns given in a `WHERE` clause. By supporting SPARQL `CONSTRUCT` queries, it allows each query to specify a complex graph structure to be returned in the query results. At the same time, the application is able to add additional functionality not supported in standard SPARQL. The Knora API server processes the input query, transforming it into one or more SPARQL queries that are executed by the triplestore, using data structures that are more complex than the ones in the input query. It then processes and transforms the triples returned by the triplestore, this time converting complex data structures into simpler ones, to construct the response.

This extra layer of processing enables Gravsearch to avoid the disadvantages of SPARQL endpoints and to provide additional features. Certain data structures can be queried in a more convenient way, 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.

2.1. Scope of Gravsearch

Gravsearch was developed as part of Knora and is used as part of the Knora API. Knora is an integrated system, taking care of creating, updating and reading data, mediating between the triplestore and the client. Gravsearch is not meant to replace an actual SPARQL endpoint, but to provide an integrated system with a flexible way of querying data. The result of a Gravsearch query is returned in the same format used by the rest of the Knora API, making it suitable for web applications.

When the full flexibility of SPAQRL is needed, projects always have the possibility to make their data openly accessible via a SPAQRL endpoint. In such a case, a named graph can be made accessible read-only and possibly transformed into a simpler data model than used internally in order to facilitate integration with other data sets.

It is true that, even with pagination, it is possible to write 'inefficient or complex SPARQL that returns only a few results' [17], and this could also be true of Gravsearch queries. As a last resort, some triplestores provide a way to set arbitrary limits on execution time or the number of triples or rows returned, but this still means consuming significant resources before rejecting the query. Moreover, setting a limit on the number of triples returned by a `CONSTRUCT` query can cause the triplestore to return incomplete entities, with no indication to the client that the response has been truncated.

It would be better to reject a badly written query without running it, and users would appreciate error messages explaining what needs to be changed to improve the query. The presence of an application layer that analyses and transforms the input query provides opportunities to do this. This is why, for example, Gravsearch currently does not allow subqueries. It would also be possible to reject a triple pattern like `?s ?p ?o` whose variables are not restricted by any other pattern. We see this kind of analysis as a promising topic for future research.

Although SELECT queries are not currently 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.

2.2. 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, thus hiding some complexity from the user. 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 dcterms, in which values are typically 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`.

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.

To illustrate the differences between these schemas, Appendix A shows a Gravsearch query that searches for a letter in the Bernoulli-Euler Online (BEOL) project (see Section 3). The query searches for a letter from the mathematician Anders Johan Lexell (identified by his IRI), specifying that the text of the letter must refer to a person whose birthdate is after 1706 CE. To do this, the query in Listing 1 searches the triples representing the text's standoff markup (see

Section 3.3) for a link to a `beol:person` resource whose birthdate is greater than that date.

To run this query, the Gravsearch transformer generates a SPARQL `CONSTRUCT` query; part of the triplestore's response is shown in Listing 2 (the full response contains many more details such as permissions, timestamps, and so on). A `beol:person` representing Leonhard Euler is returned, because he is identified in the text markup and was born after 1706. Note that the date values are represented as ranges of JDNs, and each markup tag is represented as an RDF entity.

If the client requested a JSON-LD response in the complex schema, it will look like Listing 3. Here each date is represented as a calendar date, the text with its markup has been converted to an XML document (one of several possible representations that the client can request), and each link to a `beol:person` resource is represented as a JSON-LD object containing the target resource as a nested object. Each resource is accompanied by additional metadata added by the Knora API server (some of which is not shown here), e.g. an ARK URL serving as a permanent link to the resource.

This JSON-LD representation in the complex schema is designed to be convenient to use in the development of web applications. The ARK URLs in this listing can be opened in a web browser to show an example of such an application.[13]

If the client requested a JSON-LD response in the simple schema, it will look like Listing 4. (The client can also request an equivalent response in Turtle or RDF/XML.) Here, the structure of the response has been simplified, so that values are represented as simple literals. For example, the object of `beol:title` is a string literal rather than a `TextValue`. Moreover, if the client dereferences the BEOL ontology IRI given in the JSON-LD context,[14] the definition of `beol:title` says that it is a datatype property, and that it is a subproperty of `dcterms:title`. A client that knows about `dcterms:title`, and processes titles in some particular way, can then apply the same processing to the `beol:title`. As this example shows, the simple schema lends itself to use in applications that rely mainly on standard ontologies to integrate data from different sources.

[13]For example, see http://ark.dasch.swiss/ark:/72163/1/0801/25Ro4c5gSIioLstzJ25FiAC for a representation of the letter.

[14]http://api.dasch.swiss/ontology/0801/beol/simple/v2

The simple schema also makes it possible to use standard ontologies in the Gravsearch query itself. Listing 5 shows a Gravsearch query that uses the FOAF ontology to search the BEOL project data for `foaf:Person` objects whose `foaf:familyName` is 'Euler', without using the BEOL ontology at all. This works because `beol:person` is a subclass of `foaf:Person`, and `beol:hasFamilyName` is a subproperty of `foaf:familyName`. Depending on the triplestore's inference capabilities, Knora either uses the triplestore's RDFS inference or expands the query itself using property path syntax.

The query in Listing 5 specifies that the object of `foaf:familyName` is an `xsd:string`. This is consistent with the definition of the subproperty `beol:hasFamilyName` in the simple schema. In the internal schema, the object of that property is the IRI of a Knora `TextValue`, which contains a string as well as permissions and other metadata. When Gravsearch generates SPARQL from the input query, it transforms the `WHERE` clause of the input query to handle this difference (see Section 2.6). This design allows standard ontologies using simple literal values, such as FOAF, to be used in Gravsearch queries, even though Knora actually stores values as IRIs with attached metadata.

In short, Gravsearch transforms queries, ontologies, and data on the fly according to the ontology schema that the client is using. The implementation is partly object-oriented (the Scala classes representing different sorts of RDF entities have methods for generating representations of themselves in different schemas) and partly based on sets of transformation rules for each schema (e.g. to remove certain properties and add others, and to change OWL cardinalities). Additional external schemas could be added in future.

2.3. Permissions

In Knora, each resource and each value has role-based 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:

```
<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.

2.4. 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.

Gravsearch is designed to 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 use `previousVersion` to query a past version of a value, which would be possible if they could submit SPARQL directly to the triplestore. A specific (non-Gravsearch) API request enables clients to access the version history of a value. The version history data structure is hidden from clients, enabling it to be changed in the future as RDF technology develops.

2.5. 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, and does not allow `LIMIT`. 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. (This is different from the standard SPARQL `OFFSET`, which counts solutions to the pattern in the `WHERE` clause, rather than pages of results.) The number of results per page is configurable in the application's settings, and cannot be controlled by the client.

2.6. Processing and execution of a Gravsearch query

In processing Gravsearch queries, the API server is free to use a SPARQL design that best suits the performance characteristics of the triplestore. For example, as described below, our implementation transforms each input query into multiple SPARQL queries that are run in the triplestore, and generates different SPARQL for different triplestores. Clients and users need not be aware of this.

In theory, such transformations could be implemented using a more generic rule system, such as Rule Interchange Format (RIF),[15] which is intended to be used for this purpose by the GeoSPARQL standard.[16] However, RIF is not widely implemented, and even most GeoSPARQL implementations do not support query rewriting [21].[17]

Rather than introduce an additional programming language for query transformation, we chose to implement a traditional compiler design in Scala, taking advantage of the RDF4J[18] SPARQL parser and Knora's extensive built-in support for working with OWL ontologies. An abstract syntax tree (AST) is constructed,

[15] https://www.w3.org/TR/rif-overview/

[16] https://www.opengeospatial.org/standards/geosparql

[17] Another example of a mapping language for RDF is R2RML, which intended for defining mappings between relational databases and RDF datasets. Similarly, EPNet [22] maps SPARQL queries onto heterogeneous queries in different sorts of databases.

[18] https://rdf4j.org

type information is collected from it (see 2.6.3), and it is passed through a sequence of transformations, each of which recursively traverses the AST, transforming graph patterns to produce a new AST. There are different backends for different triplestores, which take into account triplestore-specific features such as RDFS inference capabilities and full-text search. We are currently exploring adding an optimisation phase to reorder patterns in the `WHERE` clause in cases where the triplestore's query optimiser may not do so effectively.

For the reasons explained in Section 1.2, the generated SPARQL is considerably more complex than the provided Gravsearch query, and deals with data structures in the internal schema. Each Gravsearch query is converted to two SPARQL queries to improve performance. First, a `SELECT` query (prequery) is generated, to identify a page of matching resources. Then a `CONSTRUCT` query (main query) is generated, to retrieve the requested values of those resources.

2.6.1. Generation of the Prequery

The prequery's purpose is to get one page of IRIs of matching resources and values. It is a `SELECT` query whose `WHERE` clause is generated by transforming the statements of the input query's `WHERE` clause. Thus it contains all the restrictions of the input query, transformed into the internal schema (see 2.2), as well as additional statements, e.g. to implement RDFS inference if the triplestore's inference capabilities are not adequate, and to ignore resources and values that have been marked as deleted. The prequery's `SELECT` clause is generated automatically.

The prequery's result consists of the IRIs of one page of matching main resources, along with their values and linked resources. Since correct paging requires the query to return one row per matching main resource, the results are grouped by main resource IRI, and the IRIs of matching values and of linked resources are aggregated using `GROUP_CONCAT`. The results are ordered by the criteria in the `ORDER BY` clause of the input query, as well as by resource IRI (to ensure deterministic results).

The following sample Gravsearch query gets all the entries with sequence number equal to 10 from different manuscripts.

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

```
CONSTRUCT {
 ?entry knora-api:isMainResource true .
 ?entry beol:manuscriptEntryOf ?manuscript .
 ?entry beol:seqnum ?seqnum .
} WHERE {
 ?entry a beol:manuscriptEntry .
 ?entry beol:manuscriptEntryOf
   ?manuscript .
 ?manuscript a beol:manuscript .
 ?entry beol:seqnum ?seqnum .
 FILTER(?seqnum = 10)
}
```

The resulting prequery looks like this:

```
SELECT DISTINCT
 ?entry
 (GROUP_CONCAT(DISTINCT(?manuscript);
   SEPARATOR='\u001F')
   AS ?manuscript__Concat)
 (GROUP_CONCAT(DISTINCT(?seqnum);
   SEPARATOR='\u001F')
   AS ?seqnum__Concat)
 (GROUP_CONCAT(DISTINCT(
   ?manuscriptEntryOf__LinkValue);
   SEPARATOR='\u001F')
   AS
   ?manuscriptEntryOf__LinkValue__Concat)
 WHERE {...}
 GROUP BY ?entry
 ORDER BY ASC(?entry)
 LIMIT 25
```

The variables in the prequery's `SELECT` clause represent the IRIs of the matching main resources and the values present in the input query's `WHERE` clause. Besides the IRIs of the main resources, all IRIs are returned concatenated as the result of an aggregation function.

2.6.2. Generation of the Main Query

The result of the prequery is a collection of IRIs, organised by the matching main resources. Since the prequery's `SELECT` clause was generated, the variables can now be accessed in the results and further processed automatically to generate the main query, a SPARQL `CONSTRUCT` query. The prequery identified the matching resources, but it neither returned any further information other than the IRIs nor did it perform permission checking.

To generate the main query the results of the prequery are used. The concatenated IRIs are split and turned into a collection. The main query specifically

asks for the resources and values whose IRIs have been returned by the prequery.

Other than the prequery, the main query's WHERE clause is detached from the input query. The main query's structure is always the same. The application creates statements using the VALUES keyword and inserts the IRIs returned by the prequery for different categories of information combined as UNION blocks: the main resources, linked resources, values, and standoff (if necessary).

The code below shows a snippet from the main query.

```
CONSTRUCT {
  ...
}
WHERE {
 {
   VALUES ?mainResourceVar {
    <http://rdfh.ch/0803/1>
     <http://rdfh.ch/0803/2> ...
   }
 ?mainResourceVar a knora-base:Resource .
    ...
 }
 } UNION { ...
```

The code sample shows how information about specific resources is requested using the VALUES keyword. The main query's results contain information about all resources and values contained in the input query's WHERE clause. The application can now perform permission checks and filter out those resources and values the client has insufficient permissions to see. Also the application only returns the resources and values the client specified in the input query's CONSTRUCT clause by removing the information from the results the client does not want so see in the results.

As a last step, the application organises the main query's results by the main resource, returning a JSON-LD array with an element per main resource.

2.6.3. 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 requires the types of the entities used in a query so it can generate the correct SPARQL. Specifically, if a query uses the simple schema, it needs to be expanded to work with the internal schema, by taking into account an additional layer of value entities rather than simple literal values. The compiler therefore needs to know:

- The type of each variable or entity IRI used as the subject or object of a statement.
- The type that is expected as the object of each property used.

For the sake of efficiency, it is desirable to obtain this information without doing additional SPARQL queries, using only the information provided in the query itself along with the available ontologies in the triplestore (which Knora keeps in memory).[19]

Gravsearch therefore implements a simple type inference algorithm, focusing on identifying the types that are relevant to the compiler.[20] It first collects the set of all entities whose types need to be determined (*typeable entities*) in the WHERE clause of the input query. It then runs the WHERE clause through a pipeline of *type inspectors*. Each inspector implements a particular mechanism for identifying types, and passes an intermediate result to the next inspector in the pipeline.

At the end of the pipeline, each typeable entity should have exactly one type from the set of types that are relevant to the compiler. 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 in that set), it returns an error message rather than an empty response.

The first type inspector reads type annotations. Two annotations are supported:

- `rdf:type`, used to specify the types of entities that are used as subjects or objects.
- `knora-api:objectType`, which can be used to specify the expected type of the object of a non-Knora property such as `dcterms:title`.

Annotations are needed when a query uses a non-Knora ontology such as FOAF or dcterms. In the future, it would be useful for Knora to take type information from standard non-Knora ontologies, making these annotations unnecessary.

The second inspector in the pipeline infers types by using class and property definitions in ontologies. It

[19] This approach contrasts with a mechanism such as SHACL [23], which can run SPARQL queries in the triplestore.

[20] The Knora base ontology determines the set of types that the algorithm needs to identify, and thus simplifies the algorithm. For an attempt at more complete type inference for SPARQL queries, see [24].

runs each typeable entity through a pipeline of inference rules. These include rules such as the following:

- The type of a property's object is inferred from the expected object type of the property (which is specified in the definition of each Knora property).
- The expected object type of a property is inferred from the type of its actual object. (Thus if the query specifies `?book dcterms:title ?title` and `?title a xsd:string`, the rule infers that `dcterms:title` expects an `xsd:string` object.)
- If a `FILTER` expression compares two entities, they are inferred to have the same type.
- Function arguments are inferred to have the required types for the function.

Since the output of one rule may allow another rule to infer additional information, the pipeline of inference rules is run repeatedly until no new information can be determined. In practice, two iterations are sufficient for most realistic queries.

Appendix B 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.

3. Use case from the Bernoulli-Euler Online project

One project that is using Gravsearch is Bernoulli-Euler Online (BEOL),[21] 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.

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 C 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

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

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 E).

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 D).

In generating SPARQL to perform the requested search, the Gravsearch compiler converts the date comparison to one that uses a JDN. 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)
```

Gravsearch 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)
```

Other date comparisons work as follows:

- Two `DateValue` objects are considered equal if there is any overlap between their date ranges.
- Two `DateValue` objects are considered unequal if there is no overlap between their date ranges.
- `DateValue` A is considered to be less than `DateValue` B if A's end date is less than B's start date.

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")
```

Gravsearch converts this function to triplestore-specific SPARQL that (unlike the standard SPARQL `CONTAINS` function) 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 text containing the word *Acta* that is marked up as a bibliographical reference.[22]

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 straightforward to represent non-hierarchical structures in markup,[23] and 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. Projects may define own standoff entities in their project-specific

[22] 'Acta' refers to an article published by Jacob Bernoulli in *Acta Eruditorum* in December 1695.

[23] See the TEI guidelines for a discussion of this problem.

ontologies, deriving them from the types defined by the Knora base ontology.

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
  ?standoffBibliographyTag .
?standoffBibliographyTag a
  beol:StandoffBibliographyTag .
FILTER knora-api:matchInStandoff(
  ?textStr,
  ?standoffBibliographyTag,
  "Acta")
```

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[24]:

```
?textStr lucene:fullTextSearchIndex
  "Acta"^^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,
  "Acta", "i"))
```

4. Conclusion

To ensure the long-term accessibility of research data, a case can be made for storing nearly all data (with the exception of a few binary file formats) in RDF, in a way that works with any standards-compliant triplestore, and avoiding non-standard technologies and vendor lock-in. However, this approach implies that clients cannot be given direct access to the triplestore, both because the necessary generic data structures are inconvenient for clients to use, and because features such as versioning and access control cannot be handled by the triplestore in a standard way. There are also scalability issues associated with using a SPARQL endpoint backed directly by the triplestore. Yet humanities researchers want to be able to do the sorts of powerful graph searches that they can do in SPARQL.

The solution proposed here is to let users submit SPARQL, but to an API server rather than to the triplestore. This allows the API server to better 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. In short, this approach allows us to store data in a way that is suitable for long-term preservation, while querying and serving it in a way that is more appropriate for web application development. Thus it contributes to two common goals in digital humanities: making data accessible and interoperable while also ensuring its longevity.

Acknowledgements

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

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

Appendix A. Ontology schemas

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

CONSTRUCT {
  ?letter knora-api:isMainResource true .
  ?letter beol:title ?title .
  ?letter beol:hasAuthor <http://rdfh.ch/0801/1PkbSo2nRIeOK9ISp6JSdg> .
  ?letter beol:hasRecipient ?recipient .
  ?letter beol:creationDate ?creationDate .
  ?letter beol:hasText ?text .
  ?letter knora-api:hasStandoffLinkTo ?person .
  ?person beol:hasBirthDate ?birthDate .
} WHERE {
  ?letter a beol:letter .
  ?letter beol:title ?title .
  ?letter beol:hasAuthor <http://rdfh.ch/0801/1PkbSo2nRIeOK9ISp6JSdg> .
  ?letter beol:hasRecipient ?recipient .
  ?letter beol:creationDate ?creationDate .
  ?letter beol:hasText ?text .
  ?text knora-api:textValueHasStandoff ?entityTag .
  ?entityTag a beol:StandoffEntityTag .
  FILTER knora-api:standoffLink(?letter, ?entityTag, ?person)
  ?person a beol:person .
  ?person beol:hasBirthDate ?birthDate .
  FILTER(knora-api:toSimpleDate(?birthDate) >= "GREGORIAN:1706 CE"^^knora-api-simple:Date)
  ?letter knora-api:hasStandoffLinkTo ?person .
}
```

Listing 1: Searching for a letter whose text mentions a person born after 1706

```
@prefix knora-base: <http://www.knora.org/ontology/knora-base#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix beol: <http://www.knora.org/ontology/0801/beol#>

<http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA> a beol:letter;
  rdfs:label "Lexell à Condorcet";
  beol:title <http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/LWf9YstAQ4iq4CI1MmOzUA>;
  beol:creationDate <http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/M4eNIJ1MR_eB9mvTGtaZng>;
  beol:hasAuthorValue <http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/7ExuBcDbQZyRJRbAcORyKg>;
  beol:hasRecipientValue <http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/wCaspGWvS6SZXeuZy3h9MA>;
  beol:hasText <http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/im2jaDG_RKCx999ozN3RZg> .

<http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/LWf9YstAQ4iq4CI1MmOzUA> a knora-base:TextValue;
  knora-base:valueHasString "Annexe 1: Lexell à Condorcet, Saint-Pétersbourg, 2 (13) décembre 1775" .

<http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/M4eNIJ1MR_eB9mvTGtaZng> a knora-base:DateValue;
  knora-base:valueHasCalendar "JULIAN";
  knora-base:valueHasEndJDN 2369712;
  knora-base:valueHasEndPrecision "DAY";
  knora-base:valueHasStartJDN 2369712;
  knora-base:valueHasStartPrecision "DAY" .

<http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/7ExuBcDbQZyRJRbAcORyKg> a knora-base:LinkValue;
```

```
  rdf:object <http://rdfh.ch/0801/1PkbSo2nRIeOK9ISp6JSdg>;
  rdf:predicate beol:hasAuthor;
  rdf:subject <http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA> .

<http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/wCaspGWvS6SZXeuZy3h9MA> a knora-base:LinkValue;
  rdf:object <http://rdfh.ch/0801/7z7sHo5aTLqSUeSXR14qpg>;
  rdf:predicate beol:hasRecipient;
  rdf:subject <http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA> .

<http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/im2jaDG_RKCx999ozN3RZg> a knora-base:TextValue;
  knora-base:valueHasStandoff
    <http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/im2jaDG_RKCx999ozN3RZg/standoff/3>;
  knora-base:valueHasString """
            Monsieur le Marquis
            Ayant communiqué à Monsieur Euler [...]""" .

<http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/im2jaDG_RKCx999ozN3RZg/standoff/3>
  a beol:StandoffEntityTag;
  beol:standoffEntityType "person";
  knora-base:standoffTagHasEnd 90;
  knora-base:standoffTagHasLink <http://rdfh.ch/0801/NbmhfOBlQoGbX3HwXuC3Tg>;
  knora-base:standoffTagHasStart 90 .

<http://rdfh.ch/0801/7z7sHo5aTLqSUeSXR14qpg> a beol:person;
  rdfs:label "Le marquis de Condorcet" .

<http://rdfh.ch/0801/1PkbSo2nRIeOK9ISp6JSdg> a beol:person;
  rdfs:label "Anders Johan Lexell" .

<http://rdfh.ch/0801/NbmhfOBlQoGbX3HwXuC3Tg> a beol:person;
  beol:hasBirthDate <http://rdfh.ch/0801/NbmhfOBlQoGbX3HwXuC3Tg/values/vtlhBetvSRONLutapTKcaw>;
  rdfs:label "Leonhard Euler" .

<http://rdfh.ch/0801/NbmhfOBlQoGbX3HwXuC3Tg/values/vtlhBetvSRONLutapTKcaw> a knora-base:DateValue;
  knora-base:valueHasCalendar "GREGORIAN";
  knora-base:valueHasEndJDN 2344633;
  knora-base:valueHasEndPrecision "DAY";
  knora-base:valueHasStartJDN 2344633;
  knora-base:valueHasStartPrecision "DAY";
  knora-base:valueHasString "1707-04-15 CE" .
```

Listing 2: Query results in the internal schema

```
{
  "@id": "http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA",
  "@type": "beol:letter",
  "beol:creationDate": {
    "@id": "http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/M4eNIJ1MR_eB9mvTGtaZng",
    "@type": "knora-api:DateValue",
    "knora-api:dateValueHasCalendar": "JULIAN",
    "knora-api:dateValueHasEndDay": 2,
    "knora-api:dateValueHasEndEra": "CE",
    "knora-api:dateValueHasEndMonth": 12,
    "knora-api:dateValueHasEndYear": 1775,
    "knora-api:dateValueHasStartDay": 2,
    "knora-api:dateValueHasStartEra": "CE",
    "knora-api:dateValueHasStartMonth": 12,
    "knora-api:dateValueHasStartYear": 1775,
    "knora-api:valueAsString": "JULIAN:1775-12-02 CE"
```

```
    },
    "beol:hasAuthorValue": {
      "@id": "http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/7ExuBcDbQZyRJRbAcORyKg",
      "@type": "knora-api:LinkValue",
      "knora-api:linkValueHasTarget": {
        "@id": "http://rdfh.ch/0801/1PkbSo2nRIeOK9ISp6JSdg",
        "@type": "beol:person",
        "knora-api:arkUrl": {
          "@type": "xsd:anyURI",
          "@value": "http://ark.dasch.swiss/ark:/72163/1/0801/1PkbSo2nRIeOK9ISp6JSdgr"
        },
        "rdfs:label": "Anders Johan Lexell"
      }
    },
    "beol:hasRecipientValue": {
      "@id": "http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/wCaspGWvS6SZXeuZy3h9MA",
      "@type": "knora-api:LinkValue",
      "knora-api:linkValueHasTarget": {
        "@id": "http://rdfh.ch/0801/7z7sHo5aTLqSUeSXR14qpg",
        "@type": "beol:person",
        "knora-api:arkUrl": {
          "@type": "xsd:anyURI",
          "@value": "http://ark.dasch.swiss/ark:/72163/1/0801/7z7sHo5aTLqSUeSXR14qpgJ"
        },
        "rdfs:label": "Le marquis de Condorcet"
      }
    },
    "beol:hasText": {
      "@id": "http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/im2jaDG_RKCx999ozN3RZg",
      "@type": "knora-api:TextValue"
      },
      "knora-api:textValueAsXml": "[...] Monsieur le Marquis [...]",
    },
    "beol:title": {
      "@id": "http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/LWf9YstAQ4iq4CI1MmOzUA",
      "@type": "knora-api:TextValue"
      },
      "knora-api:valueAsString": "Annexe 1: Lexell à Condorcet, Saint-Pétersbourg, 2 (13) décembre 1775"
    },
    "knora-api:arkUrl": {
      "@type": "xsd:anyURI",
      "@value": "http://ark.dasch.swiss/ark:/72163/1/0801/25Ro4c5gSIioLstzJ25FiAC"
    },
    "knora-api:hasStandoffLinkToValue": {
      "@id": "http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA/values/aK3JZSkOR6StGyiaIn6Vlg",
      "@type": "knora-api:LinkValue",
      "knora-api:linkValueHasTarget": {
        "@id": "http://rdfh.ch/0801/NbmhfOBlQoGbX3HwXuC3Tg",
        "@type": "beol:person",
        "beol:hasBirthDate": {
          "@id": "http://rdfh.ch/0801/NbmhfOBlQoGbX3HwXuC3Tg/values/vtlhBetvSRONLutapTKcaw",
          "@type": "knora-api:DateValue",
          "knora-api:dateValueHasCalendar": "GREGORIAN",
          "knora-api:dateValueHasEndDay": 15,
          "knora-api:dateValueHasEndEra": "CE",
          "knora-api:dateValueHasEndMonth": 4,
          "knora-api:dateValueHasEndYear": 1707,
          "knora-api:dateValueHasStartDay": 15,
          "knora-api:dateValueHasStartEra": "CE",
          "knora-api:dateValueHasStartMonth": 4,
          "knora-api:dateValueHasStartYear": 1707,
```

```
        "knora-api:valueAsString": "GREGORIAN:1707-04-15 CE"
      },
      "knora-api:arkUrl": {
        "@type": "xsd:anyURI",
        "@value": "http://ark.dasch.swiss/ark:/72163/1/0801/NbmhfOBlQoGbX3HwXuC3Tg_"
      },
      "rdfs:label": "Leonhard Euler"
    }
  },
  "rdfs:label": "Lexell à Condorcet",
  "@context": {
    "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "knora-api": "http://api.knora.org/ontology/knora-api/v2#",
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "beol": "http://api.dasch.swiss/ontology/0801/beol/v2#",
    "xsd": "http://www.w3.org/2001/XMLSchema#"
  }
}
```

Listing 3: Gravsearch response in the complex schema

```
{
  "@id": "http://rdfh.ch/0801/25Ro4c5gSIioLstzJ25FiA",
  "@type": "beol:letter",
  "beol:creationDate": {
    "@type": "knora-api:Date",
    "@value": "JULIAN:1775-12-02 CE"
  },
  "beol:hasAuthor": {
    "@id": "http://rdfh.ch/0801/1PkbSo2nRIeOK9ISp6JSdg"
  },
  "beol:hasRecipient": {
    "@id": "http://rdfh.ch/0801/7z7sHo5aTLqSUeSXR14qpg"
  },
  "beol:hasText": "\n Monsieur le Marquis\n Ayant communiqué à Monsieur Euler [...]",
  "beol:title": "Annexe 1: Lexell à Condorcet, Saint-Pétersbourg, 2 (13) décembre 1775",
  "knora-api:arkUrl": {
    "@type": "xsd:anyURI",
    "@value": "http://ark.dasch.swiss/ark:/72163/1/0801/25Ro4c5gSIioLstzJ25FiAC"
  },
  "knora-api:hasStandoffLinkTo": {
      "@id": "http://rdfh.ch/0801/NbmhfOBlQoGbX3HwXuC3Tg"
  },
  "rdfs:label": "Lexell à Condorcet",
  "@context": {
    "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "knora-api": "http://api.knora.org/ontology/knora-api/simple/v2#",
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "beol": "http://api.dasch.swiss/ontology/0801/beol/simple/v2#",
    "xsd": "http://www.w3.org/2001/XMLSchema#"
  }
}
```

Listing 4: Gravsearch response in the simple schema

```
PREFIX knora-api: <http://api.knora.org/ontology/knora-api/simple/v2#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
CONSTRUCT {
  ?person knora-api:isMainResource true .
  ?person foaf:familyName ?familyName .
  ?person foaf:givenName ?givenName .
} WHERE {
  ?person a knora-api:Resource .
  ?person a foaf:Person .
  ?person foaf:familyName ?familyName .
  ?familyName a xsd:string .
  ?person foaf:givenName ?givenName .
  ?givenName a xsd:string .
  FILTER(?familyName = "Euler")
}
```

Listing 5: Gravsearch query using FOAF

Appendix B. 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 6: A Gravsearch template

Appendix C. 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 7: A Gravsearch template

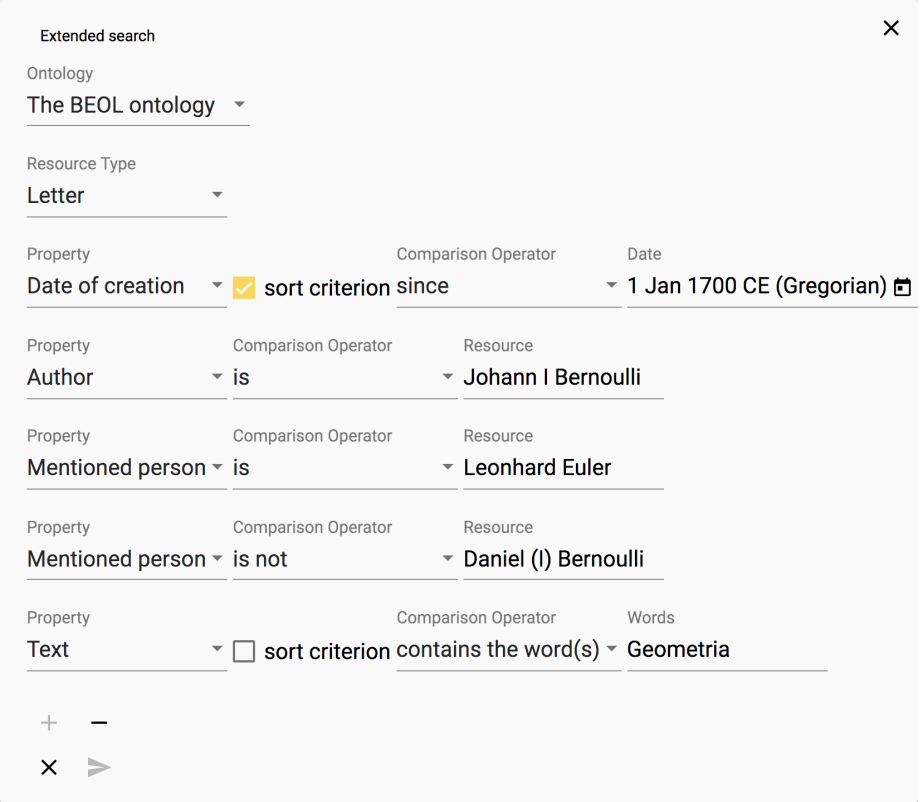
Appendix D. A user-generated query

```
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 8: A user-generated query

Appendix E. GUI widget

Figure 1. Advanced Search Widget



Appendix F. 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 9: A user-generated query

References

[1] Bibliothèque nationale de France, Virtuoso SPARQL Query Editor. https://data.bnf.fr/current/sparql.html.

[2] Isidore, Virtuoso SPARQL Query Editor. https://isidore.science/sparql.
[3] C. Allocca, A. Adamou, M. d'Aquin and E. Motta, SPARQL Query Recommendations by Example, in: *13th ESWC 2016*, 2016. http://oro.open.ac.uk/46757/.
[4] O. Bruneau, E. Gaillard, N. Lasolle, J. Lieber, E. Nauer and J. Reynaud, A SPARQL Query Transformation Rule Language: Application to Retrieval and Adaptation in Case-Based Reasoning, in: *ICCBR 2017: Case-Based Reasoning Research and Development, 25th International Conference on Case-Based Reasoning*, 2017. https://hal.inria.fr/hal-01661651.
[5] 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.
[6] S. Kirrane, A. Mileo and S. Decker, Access Control and the Resource Description Framework: A Survey, *Semantic Web* **0** (2016), 1–42. http://www.semantic-web-journal.net/content/access-control-and-resource-description-framework-survey-0.
[7] O. Hartig and B. Thompson, Foundations of an Alternative Approach to Reification in RDF, *CoRR* (2014). http://arxiv.org/abs/1406.3399.
[8] T. Schweizer, S. Alassi, M. Mattmüller, L. Rosenthaler and H. Harbrecht, An Interactive, Multi-layer Edition Of Jacob Bernoulli's Scientific Notebook Meditationes As Part Of Bernoulli-Euler Online, in: *Digital Humanities 2019 Conference Abstracts*, 2019. https://dev.clariah.nl/files/dh2019/boa/0115.html.
[9] D.D. McCarthy, The Julian and Modified Julian Dates, *Journal for the History of Astronomy* **29**(4) (1998), 327–330. doi:10.1177/002182869802900402.
[10] A. Grafton, Some Uses of Eclipses in Early Modern Chronology, *Journal of the History of Ideas* **64**(2) (2003), 213–229. doi:10.1353/jhi.2003.0024.
[11] GODOT, Graph of Dated Objects and Texts. https://godot.date/.
[12] chronOntology, iDAI.chronontology. https://chronontology.dainst.org.
[13] PeriodO, A gazetteer of periods for linking and visualizing data. https://perio.do.
[14] DBpedia, Virtuoso SPARQL Query Editor. http://dbpedia.org/sparql.
[15] Wikidata Query Service. https://query.wikidata.org.
[16] Europeana, Virtuoso SPARQL Query Editor. http://sparql.europeana.eu/.
[17] D. Rogers, The Enduring Myth of the SPARQL Endpoint. https://daverog.wordpress.com/2013/06/04/the-enduring-myth-of-the-sparql-endpoint/.
[18] GraphQL, GraphQL. https://www.howtographql.com/.
[19] HyperGraphQL, HyperGraphQL. http://hypergraphql.org/.
[20] S. Klarman, Querying DBpedia with GraphQL. https://medium.com/@sklarman/querying-linked-data-with-graphql-959e28aa8013.
[21] T. Ioannidis, G. Garbis, K. Kyzirakos, K. Bereta and M. Koubarakis, Evaluating Geospatial RDF stores Using the Benchmark Geographica 2, 2019. https://arxiv.org/abs/1906.01933.
[22] D. Calvanese, P. Liuzzo, A. Mosca, J. Remesal, M. Rezk and G. Rull, Ontology-based data integration in EPNet: Production and distribution of food during the Roman Empire, *Engineering Applications of Artificial Intelligence* **51** (2016), 212–229. doi:10.1016/j.engappai.2016.01.005.
[23] W3C, SHACL. https://www.w3.org/TR/shacl/.
[24] P. Seifer, M. Leinberger, R. Lämmel and S. Staab, Semantic Query Integration With Reason, *The Art, Science, and Engineering of Programming* **3**(3) (2019). doi:10.22152/programming-journal.org/2019/3/13.