Treating Query Graphs As Connecting Trees

SPARQL Queries to FHIR API

Eric Prud'hommeaux [1,2,*], Claude Nanjo[3] and Jerven Bolleman[4]

[1]Leiden University Medical Center
[2]4MedBox Europe
[3]University of Utah
[4]Swiss Institute for Bioinformatics

Abstract

FHIR[1] RDF defines the RDF representation of vast swaths of clinical data. HAPI FHIR, the Java reference implementation for FHIR, can read and write FHIR RDF using FHIR's REST-ful API, which at least in theory enables merging clinical data with the larger Semantic Web infrastructure.

There exists, however, few native SPARQL endpoints for FHIR. This paper presents an approach called Fhir-Sparql, whereby SPARQL queries can interface with the FHIR REST API using a novel intermediate structure called ArcTrees, to address a large number of primary use cases for clinical data. Unlike HAPI FHIR queries, which are primarily resource-centric, Fhir-Sparql allows for composing detailed declarative queries spanning different FHIR Resource types.

Fhir-Sparql decomposes a user query into ArcTrees, which are then easily matched against a library mapping ArcTrees to FHIR REST API operations. The paper describes that process with the intention of (attracting users and collaborators and) illustrating to the community how to methodically ensure completeness in SPARQL queries over non-RDF data.

Keywords

FHIR, SPARQL, Query projection, Clinical data

1. Introduction

Making no grand pretensions about the claim of semantics, "semantic" data is self-describing and pre-unified, meaning the identifiers within the data connect it to concepts and relations used elsewhere in the Semantic Web. Theoretically, the existence of FHIR RDF makes it possible for clinical data to participate in the Semantic Web. In practice, projecting vast swaths of data into another format incurs costs for processing, storage, security, and especially concurrency.

Like most Semant Web geeks, the authors are motivated to connect data and enable use cases that would otherwise require bespoke data pipelines and procedural code to "mash-up" data. Our specific use cases consisted of identifying cohorts of patients sharing common characteristics for various clinical or research purposes. Such patient data is often stored in repositories exposing a FHIR REST API or in relational enterprise data warehouses.

Given the heterogeneity of systems and APIs, the solution is often to rely on the query tools already supported by a data repository, such as an electronic health record system's FHIR API or a research database's OMOP [2] [3] interface. A typical (non-semantic) cohort generation tool will request and consume data in a JSON format, and procedurally iterate through that data, potentially merging it with outside data, e.g genomic data.

Yet, such an approach places a burden on a researcher conducting exploratory research. Researchers already familiar with RDF technologies will want to use SPARQL to query data. Thus, one way to

SWAT4HCLS 2025: The 16th International Conference on Semantic Web Applications and Tools for Health Care and Life Sciences, February 24–27, 2025, Barcelona, Spain

*Corresponding author.
eric@uu3.org (E. P.); claude.nanjo@utah.edu (C. Nanjo); jerven.bolleman@sib.swiss (J. Bolleman)
https://uu3.org/People/Eric/ (E. P.); https://faculty.utah.edu/u6017542-Claude_Nanjo/hm/index.hml (C. Nanjo)
http://orcid.org/0000-0003-1775-9921 (E. P.); 0009-0002-1208-8858 (C. Nanjo); 0000-0002-7449-1266 (J. Bolleman)

published 2026-04-22

reduce this cognitive burden is to recognize a set of SPARQL queries and/or patterns, execute them as repository-native queries, and project the results back as SPARQL results. While such an approach could be designed to address very specific use cases, a generalizable approach providing a sound and complete transformation allows users to perform arbitrary queries without first ensuring that the transformation tool faithfully honor the semantics and return the same results as would the same query performed over a complete projection of the native data into an RDF graph store.

The ArcTrees approach described in this paper proved to be a practical way to describe the patterns served by simple query APIs like FHIR's REST API.

2. Notation

This document assumes a familiarity with RDF and SPARQL. For the FHIR aspects, this introduction should render the rest of the document intelligible. "FHIR" denotes multiple things:

- `A Data Model` : FHIR `Resources` are a set of data structures related to patient care with defined content models. These have official serializations in XML, JSON and RDF and represent the payload of the FHIR API. A typical use case will involve `Resources` like `Observation`, `Procedure` and `Patient`.
- `A Resource-Oriented RESTful Protocol` : The FHIR REST API supports CRUD operations on a repository of FHIR `Resources` enhanced by search operations matching search arguments against specific points in the content model.
- `A Metamodel` : The FHIR StructureDefinition `Resource` is a meta-modeling construct used to define the content model of FHIR (i.e., all FHIR `Resources`). Naturally, there is a StructureDefinition for StructureDefinition.

3. State of the Art

The HAPI FHIR implementation is a popular open-source Java implementation of FHIR. It includes support for consuming and emitting RDF representations of FHIR Resources.

As far as we know, there are no tools to methodically recognize FHIR REST parameters implied by any query language.

With regard to orchestrating queries spanning multiple resources, there exist only a few technologies for joining FHIR Resources:

- `Chaining` [4] Although this could in principle address the subset of Fhir-Sparql use cases that join on attributes supported by chaining (which excludes the use case exemplified in this document), the subset of the FHIR REST API needed for routine patient care is more likely to be widely supported.
- `Clinical Quality Language (CQL), FHIR Path, and FHIR Query Language (FQL)` [5] [6] [7] Special-purpose expression and query languages for accessing clinical (for CQL) or specifically FHIR (FQL) data.
- `OMOP Common Data Model` [2] A research-oriented relational model for clinical data.

4. Architecture

In the proposed approach, ArcTrees derived from SPARQL are mapped to FHIR Search queries. We provide a very brief introduction to each of the following.

4.1. ArcTrees

ArcTrees were inspired by the fact that most SPARQL queries (and the data they match) consist of a set of trees sparsely interconnected by variables. ArcTrees are representations of these trees along with a set of variables mapped to their occurrences in those trees.

The ArcTree data structure is simply a triple pattern `tp` from the SPARQL query and a possibly empty set of outgoing ArcTrees representing the set of triple patterns having tp's object as their subject.

```
public tp: Triple,
public out: ArcTree[]
```

So far, use cases have motivated only trees in this "arcs-out" direction. It's possible that some data may motivate the extension to "arcs-in" ArcTrees at some point.

ArcTrees were developed for Fhir-Sparql because they could express both the user queries and the FHIR RDF graph structures captured by the REST API query parameter. As an example of a REST API parameter, consider a tobacco smoking status FHIR Observation Resource with LOINC code "72166-2" having a value which captures whether a patient is a current smoker, former smoker, etc... The FHIR specification defines a REST parameter `code=<myCode>` for the FHIR Observation Resource, which can be used to find `Observations` whose `fhir:code` has a `fhir:coding` of some list of `Codings`, one of which has a `fhir:code` (yes, the same predicate is used twice in FHIR RDF) identified by `<myCode>`. Here, `<myCode>` is a string with format "system|code". For instance, the LOINC code 72166-2 would be represented as follows: "http://loinc.org|72166-2" and subsequently micro-parsed into a `system` and a `code`.

```
arcTree:
  tp: [] fhir:code _:b1. out: [
    _:b1 fhir:coding _:b2. out: [
      _:b2 rdf:rest*/rdf:first _:b3. out: [
        _:b3 fhir:code _:b4. out: [
          _:b4 fhir:v ?v1. out: []
        ],
        _:b3 fhir:system _:b5. out: [
          _:b5 fhir:v ?v2. out: []
        ]
      ]
    ]
  ]
transform:
  (?v1, ?v2) => ?v2 + '|' + ?v1
```

To make life marginally more interesting, the `code=<myCode>` parameter can exclude the $'|'$ separator, in which case, it is interpreted as just a `fhir:code` value, i.e. the `fhir:system` may have any value or be missing entirely.

4.2. FHIR REST API

In this paper, we shall concern ourselves primarily with the FHIR Search API. FHIR search queries typically have the following form:

```
GET [server_repo_path]/[resource]?[parameter_name=parameter_value]&...
```

Omitting the resource in the query path will return all resources that share the given parameter list.

One can refine a search using query parameters. For instance, when searching for a patient's tobacco smoking status, one may use the following query:

```
GET http://my.fhir.server.org/fhir/Observation?code=http://loinc.org
    |72166-2
```

Searching for a hypertension diagnosis with a date of onset of 01/12/2024, one might write the following query:

```
GET http://my.fhir.server.org/fhir/Condition?code=http://snomed.info/sct
    |38341003&onset=2024-01-12
```

5. Fhir-Sparql Query Processing

At a high level,

1. Each query is partitioned into ArcTrees.
2. These ArcTrees are matched against FHIR REST query parameters against all possible FHIR Resources.
3. The REST parameters for the first Resource are aggregated and the query executed.
4. For each returned FHIR Resource, that portion of the SPARQL query is executed, producing a set of (Resource-specific) solutions (SPARQL Results).
5. For each solution, the bound variables are substituted into the ArcTrees for the next Resource.
6. The final result is the unification of all of the Resource-specific solutions.

6. Example Use Case - Scheduling Cancer Screenings

Our example user query selects patients that have a documented tobacco smoking status and had a lung cancer screening procedure:

```
PREFIX fhir: <http://hl7.org/fhir/>
PREFIX sct: <http://snomed.info/id/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?obs ?proc ?patRsrc WHERE {
  # Observation
  ?obs a fhir:Observation ;

    # Observation.code
    fhir:code [
      fhir:coding [
        rdf:rest*/rdf:first [
          fhir:code [ fhir:v "72166-2" ] ;
          fhir:system [ fhir:v "http://loinc.org"^^xsd:anyURI ]
    ] ] ] ;

      # Observation.subject
    fhir:subject [ fhir:reference [ fhir:link ?patRsrc ] ] .

  # Procedure
  ?proc a fhir:Procedure ;

    # Procedure.code
    fhir:code [
      fhir:coding [
        rdf:rest*/rdf:first [
          fhir:code [ fhir:v "724167008" ] ;
```

```
            fhir:system [fhir:v "http://snomed.info/sct"^^xsd:anyURI]
    ] ] ] ;

    # Procedure.subject
    fhir:subject [ fhir:reference [ fhir:link ?patRsrc ] ] .

 # Patient
  # Skip patient type arc to illustrate type inference.
  # Patient.id
  ?patRsrc fhir:id [ fhir:v ?patId ] .
}
```

(This query is layed out to make it readable; the algorithm has been tested on BGPs with arbitrary order.)

7. ArcTree Partitioning

When applied to the above example, the partitioning produces sets of ArcTrees for `Observations` and `Procedures`, along with the sole variable that connects them (`?patRsrc`) and the list of its occurrences in the `ArcTrees`.

Here, `Observation` has one `ArcTree`, which is normal because all of the top-level triples shared a common subject (`?obs`). The fhir-sparql-js implementation produces three `ArcTrees`:

```
ArcTrees[0]: <root> [
  ?obs rdf:type fhir:Observation .
  ?obs fhir:code ?code . [
    ?code fhir:coding ?codeList . [
      ?codeList rdf:rest*/rdf:first ?codeElt . [
        ?codeElt fhir:code ?codeCode . [
          ?codeCode fhir:v "72166-2" .
        ]
        ?codeElt fhir:system ?codingSystem . [
          ?codingSystem fhir:v "http://loinc.org"^^xsd:anyURI .
  ] ] ] ]
  ?obs fhir:subject ?subjectRef . [
    ?subjectRef fhir:reference ?subjectBNode . [
      ?subjectBNode fhir:link ?patRsrc .
] ] ]
ArcTrees[1]: ...ArcTree for Procedure elided for brevity...
ArcTrees[2]: <root> [
  ?patRsrc fhir:id ?patIdElt . [
    ?patIdElt fhir:v ?patId .
  ]
]
```

These ArcTrees, plus the connecting variables, are sufficient to efficiently navigate the original graph. For example the only connecting variable in the above ArcTrees is `patRsrc` and it appears in four places. These are expressed below with `pos` being the part of speech (subject or object), paired with a JSON path into the above ArcTrees:

```
{ patRsrc: [
    {pos: 'object', arcTree: ArcTrees[0].out[2].out[0].out[0]},
    {pos: 'object', arcTree: ArcTrees[1].out[2].out[0].out[0]},
    {pos: 'subject', arcTree: ArcTrees[2].out[0]},
```

```
    {pos: 'subject', arcTree: ArcTrees[2].out[1]},
] }
```

8. Matching REST Search API Patterns

Each ArcTree from the user query is mapped to FHIR REST API search parameters implied by that ArcTree. This matching algorithm starts with a list of possible Resource types. If the user query includes a type arc in that ArcTree, the list of Resources types has that single entry. Likewise, the schema (for this use case, the ShEx distributed in the FHIR specification) may constrain the possible types, as in this example, `?patRsrc` is limited to be the intersection of the `Resource` types allowed in `Observation` and `Procedure` subjects (`Patient`, `Group`, `Device`, `Practitioner`, `Organization` or `Location`). If the type is unconstrained, matching starts with the set of all `Resource` types.

For each possible `Resource` type (possibly all 190+ FHIR Resource types), we start with a set of candidate FHIR API query parameters associated with that `Resource` type (e.g. dosage for `Medications`, value for `Observations`, status for basically everything). This set is reduced to those parameters implied by the user ArcTree being examined by removing any that do not match a corresponding (nested) tree in the user ArcTree. For each candidate `Resource` type, the remaining set is a conjunction of REST parameters that can be composed into a GET request. If there is more than one candidate `Resource` type, these are treated as a disjunction (for instance, if the ArcTree has a both a status and a subject, candidate types would include `Observations` and `Procedures`, etc.).

9. Executing Queries

- As with any SPARQL join operation, start with a `SPARQL Results` set with one solution and no bindings.
- For each ArcTree extracted from the user query,
- – For each solution in the current results (initialized above),
  - For each REST parameter matched against the ArcTree,
    - Substitute any variables in the ArcTree that have been bound in this solution (none, on the first iteration).
    - Construct and execute a FHIR REST API query.
    - For each response, re-execute the portion of the SPARQL query related to that Resource type
      - Push each query solution into the results for the next iteration

This is effectively a join operation, which performs two tasks:

- Enforce constraints that don't have a corresponding FHIR REST expression.
- Unify variables used between ArcTrees.

Regarding the first bullet point above, a query that has a FILTER for `?patRsrc` matching some regular expression would not be expressible in the FHIR REST API. That FILTER would be ignored when constructing the FHIR REST parameters. Therefore the query results would need to be filtered to enforce that FILTER constraint.

Following is a more formal treatment of the algorithm described above:

- A is a set of "ArcTrees" (nested tree structures) in the SPARQL query.
- RAs is a list of sets of ArcTrees that fit in some FHIR Resource (as captured in the ShEx schema).
- Isolate $AR1$ the set of ArcTrees at the start of RAs; judicious selection of AR1 is an optimization.
- Match AR against a library of FHIR REST API definitions (e.g. $code =< code.coding.system > | < code.coding.code >$) to generate a FHIR GET operation GET1.

- Interpret results $GET1$ as RDF graph $G1$.
- (re-)Generate (Resource-specific) SPARQL Query $SR1$ from AR.
- RS is the SPARQL Result Set obtained by issuing $SR1$ on $G1$.
- For each result $R1$ in RS, prepare a query $AR2$ from $AR2$, the next in the list of RAs, given the bindings in $R1$ (e.g. $Patient/2$)
- rinse, lather, repeat

10. Next Steps

The authors intend to investigate the following in order to better scale the technology:

- `Automate harvesting of REST api ArcTree patterns` from the specification. The FHIR specification is distributed in a machine-readable format called StructureDefinition (essentially, a schema language with a lot of additional metadata).
- `Expand expressivity to work over SPARQL UNIONS and MINUS`. The current strategy involves transforming everything to a Disjunctive Normal Form. The down-side of DNF is that it often results in repetition of patterns that started out nicely factored in the user query. Such an optimization would require further research.
- `Leverage FHIR search chaining` where supported. In the cases where chaining captures the desired semantics, e.g. names of Patients referenced by Observations with a given code.
- `Resolve extremely slow query times` caused by a lack of subject indexes in the Comunica [8] SPARQL engine. An effort to port fhir-sparql-js (typescript) to java is underway, which would bypass the issue, though improving Comunica would of course benefit a large community.

11. Conclusions

SPARQL query mediation using ArcTrees offers the RDF research community a potentially valuable approach to querying heterogeneous data sources. In particular, leveraging such an approach against FHIR-enabled repositories could greatly facilitate biomedical research by offering clinical data in a well-known format (FHIR RDF). For instance, the ability to capture joins in a mature declarative language has proven useful in a project between Leiden University Medical Center and University Medical Center Utrecht, which selected amongst current transplant candidates those that were currently healthy enough to receive a transplant. While we have validated the feasibility of Fhir-Sparql in more limited contexts, given its use-case-driven development, more remains to be done in order to fully generalize, automate, and validate the proposed approach and make it more performant.

Acknowledgments

Thanks to DBCLS (Database Center for Life Science) https://dbcls.rois.ac.jp/index-en.html and Leiden University Medical Center https://www.lumc.nl/en/ for sponsoring this work.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] FHIR, Hl7 fhir release 5, 2023. URL: http://hl7.org/fhir/R5/documentation.html.
[2] OMOPCDM, Standardized data: The omop common data model, 2023. URL: https://www.ohdsi.org/data-standardization/.

[3] E. A. Voss, R. Makadia, A. Matcho, Q. Ma, C. Knoll, M. Schuemie, F. J. DeFalco, A. Londhe, V. Zhu, P. B. Ryan, Feasibility and utility of applications of the common data model to multiple, disparate observational health databases, Journal of the American Medical Informatics Association 22 (2015) 553–564. URL: https://doi.org/10.1093/jamia/ocu023. doi:10.1093/jamia/ocu023. arXiv:https://academic.oup.com/jamia/article-pdf/22/3/553/34145431/ocu023.pdf.
[4] CHAINING, FHIR search / chaining (chained parameters), 2024. URL: https://www.hl7.org/fhir/search.html#chaining.
[5] CQL, Clinical quality language (cql), 2024. URL: https://cql.hl7.org/.
[6] FHIRPATH, Fhirpath (normative release), 2024. URL: https://hl7.org/fhirpath/.
[7] Firely query language, 2024. URL: https://simplifier.net/docs/fql.
[8] A knowledge graph querying framework, 2024. URL: https://comunica.dev/docs/query/.

A. Online Resources

The sources for the FHIR SPARQL implementations are available in GitHub:

- FHIR SPARQL tests - https://github.com/fhircat/fhir-sparql-test
- Typescript implementation - https://github.com/fhircat/fhir-sparql-js