SPARQL federated query debugging tool

Marek Moos [a,*] and Jakub Galgonek [a]

[a] *Institute of Organic Chemistry and Biochemistry of the Czech Academy of Sciences, Czech Republic*
E-mails: marek.moos@uochb.cas.cz, jakub.galgonek@uochb.cas.cz

Abstract. Gaining insight into a complex problem often requires combining data from multiple datasets. For this reason, SPARQL query support within a federated environment is an important feature. However, several pitfalls have been encountered in practice, significantly complicating the use of SPARQL queries in such setups. These challenges include uninformative error responses, performance bottlenecks and unintended semantic changes introduced by SPARQL endpoints. To address these pitfalls, this paper introduces a newly implemented SPARQL query debugger, which is available as a web application at https://sparql-debugger.elixir-czech.cz. It has been developed for the purpose of monitoring, in real time, the execution of SPARQL queries that incorporate the service pattern. This monitoring is crucial for error detection and performance optimization. Detailed service execution data (such as SPARQL requests and responses, durations, etc.) can help identify the specific instance of a service responsible for a problem, even if it is deeply nested within the service execution tree. The tool is based on the principle of redirecting all requests to a debugging proxy server, so it can be used with all SPARQL-compliant endpoints without the need for their modification. The debugging tool presented in the paper enables the identification and resolution of issues that are otherwise difficult to address and has proven its effectiveness in practice.

Keywords: SPARQL Federated Query, debugger

1. Introduction

The Semantic Web is becoming increasingly pivotal across various fields that require managing complex and diverse datasets interoperably. One notable example is bioinformatics, where researchers focus heavily on the principles of Findable, Accessible, Interoperable, Reusable data [11] and seamless data integration, both of which are inherently supported by Semantic Web technologies. In practice, this entails publishing data in RDF, annotating it with ontologies and querying it using SPARQL [1, 3, 4, 6, 9, 10, 12]. In bioinformatics, interoperability is important because gaining insights into a biological problem often necessitates combining data from multiple research domains. Because biological datasets are typically produced by independent teams highly specialized in different fields of interest, it is important to be able to create queries that span multiple datasets.

There are several approaches to spanning multiple RDF datasets. One option is to load the selected datasets, or relevant portions of them, into a single triplestore, making it possible, for example, to transform and augment the original data. Another approach is to transparently split a query into subqueries and dynamically identify the appropriate endpoints for each subquery, as implemented by tools such as FedX [8] or Comunica [16]. Alternatively, subqueries can be explicitly directed to specific endpoints using the standard SPARQL Federated Query extension [20]. The advantages of this approach are that it does not require the use of any additional software and that it provides query developers with a complete overview over query federation. The work presented here focuses on the latter approach and federated query refers exclusively to the use of the service pattern within a SPARQL query.

*Corresponding author. E-mail: marek.moos@uochb.cas.cz.

In the SPARQL Federated Query extension, target endpoints are explicitly denoted by service patterns. Note that these patterns can be nested, resulting in a federated query that can be represented as a *service pattern tree*. From an operational perspective, as services are executed by individual endpoints, the evaluation of such a query effectively constitutes a *service execution tree*. Note that the two trees may differ in general, even though requests to evaluate service patterns are always initiated by endpoints that evaluate their direct parent patterns.

In practice, several pitfalls have been encountered that significantly complicate the use of federated SPARQL queries. First of all, in the event of an error, error messages from nested services are often not propagated to the top level and are effectively swallowed, resulting in uninformative query error responses. Furthermore, when a query takes an unusually long time to execute, it is not clear what the cause is, leaving users without an understanding of the problem. Last but not least, some SPARQL endpoints may silently modify subqueries they delegate to other service endpoints, so these endpoints might then interpret the modified subqueries differently, potentially leading to unexpected results.

To overcome these pitfalls, the debugging tool presented here has been developed to monitor the entire service execution tree of a federated query. The ability to monitor federated queries is crucial for both error detection and performance optimization. Detailed execution data can help identify the specific service pattern responsible for an error, even if it is nested deep within the service execution tree. Moreover, tracing can reveal service patterns that suffer from high latency or are executed too many times. This is often related to execution strategies employed by SPARQL endpoints. For instance, using the *nested loop join* [2] strategy, a specific service pattern may be executed multiple times with different substitutions of its variables based on values computed beforehand, which can result in an enormous number of remote requests. By contrast, resolving a service pattern in its original form can lead to excessively large responses. By pinpointing these bottlenecks, users can optimize their queries for better performance.

Although general monitoring platforms, such as Datadog [14], offer alternatives to our debugging tool, they require configuration and deployment on service endpoints, which may not always be feasible. By contrast, tools such as Virtuoso [18] and Jena [13] offer detailed information on executions of directly nested services, but they cannot debug the execution within those services. Therefore, despite the importance of this functionality, the tool presented here is, to the best of our knowledge, the first to provide comprehensive tracing across all levels of a service execution tree, regardless of which SPARQL engine is used at each endpoint.

2. Implementation

The presented tool is provided as a web application [1] intended for SPARQL query developers, designed to encapsulate federated query debugging within an intuitive interface. The application features a custom YASGUI [5] component for query editing, allowing users to work on multiple queries simultaneously using integrated tabs. The debugging process is initiated by pressing the `Debug` button. As services are executed by endpoints, the service execution tree is rendered continuously, showing the progress in real time. For each service call, trace information, including the state, HTTP status, request and response data, duration and number of solutions, is displayed. If a service pattern is invoked multiple times with, for example, varying variable substitutions, these calls are aggregated into a special *bulk execution node*. These nodes, highlighted in yellow, are collapsed by default to enhance clarity and include information about the total number of calls and their combined duration. It is also possible to run a query directly without debugging by triggering the `Run` button. Furthermore, both the debugging process and the query execution without debugging can be terminated using the `Cancel` button. In the case of debugging, this can save significant endpoint resources because the expansion of the service execution tree (i.e. the calling of remote services) is halted. In addition, the application also provides a selection of federated query examples.

[1] https://sparql-debugger.elixir-czech.cz/

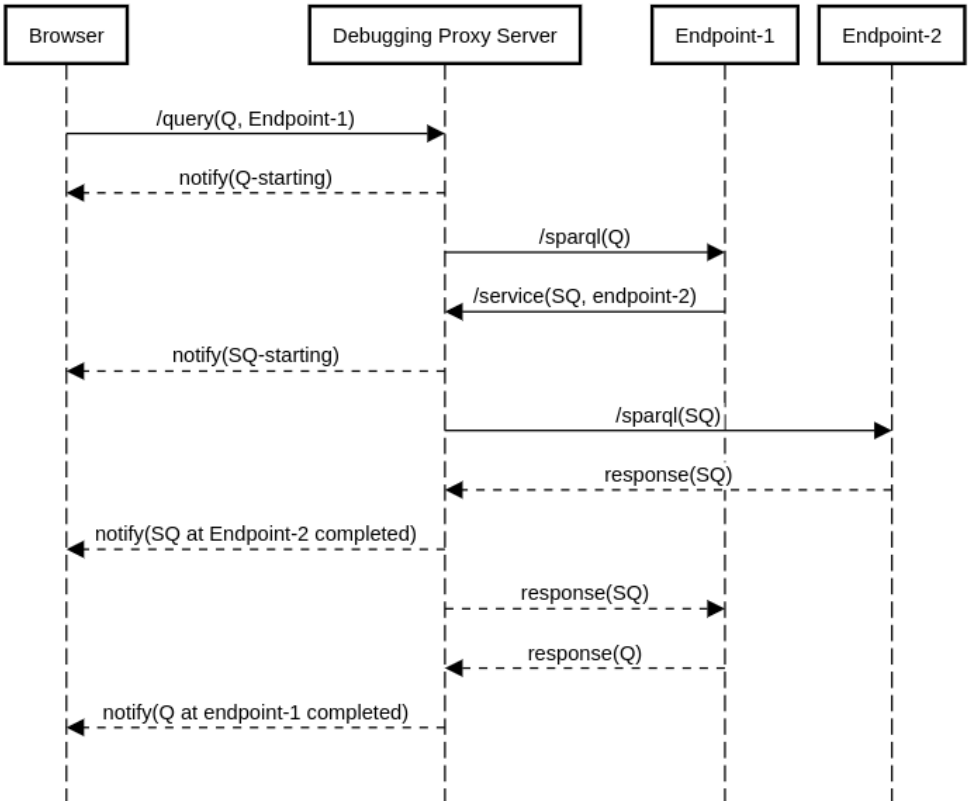

Fig. 1. Debugging proxy server

2.1. Concept

It is generally assumed that SPARQL developers cannot feasibly modify service endpoints (e.g. configure them or install additional extensions) and that their interaction with endpoints is limited only to querying via the SPARQL protocol. The debugging tool presented here is therefore implemented with a proxy server at its core. This server intercepts and wraps the execution of each service together with detailed trace information such as the request data, response data, status, etc. When the debugging process is initiated, a query is sent to the proxy server, this execution acts as the root of the entire service execution tree.

An example of query debugging is illustrated in Fig. 1. In this example, a query, denoted `Q`, is evaluated at `endpoint-1` and contains one service pattern `SQ` evaluated at `endpoint-2`. All evaluations are performed through the proxy server.

Because all service endpoints are treated as black boxes, interaction with them is only possible through SPARQL requests and responses. To properly trace services, the original service endpoint URLs in the SPARQL query request are substituted with the proxy server's URL. Additionally, essential information must be encoded into these URLs to ensure that, when services are intercepted by the proxy server, they can be properly traced and executed. The URL of the original service endpoint has to be encoded, allowing the proxy server to know which actual service endpoint to call. Likewise, the ID of its parent in the service execution tree needs to be encoded to identify where the new service execution node should be added in the tree. Lastly, the encoded query ID retains information about the query scope.

To demonstrate the debugging process on a practical example, a federated SPARQL query from the BioSODA website[15], which retrieves *genes that are orthologs of a gene expressed in the fruit fly brain*, is used.

The corresponding query service pattern tree and service execution tree are shown in Fig. 3 and Fig. 4.

```
SELECT DISTINCT ?id ?OMA_LINK WHERE {
  SERVICE <https://www.bgee.org/sparql/> {
    SELECT DISTINCT ?gene ?id {
      ?gene a orth:Gene .
      ?gene genex:isExpressedIn ?anat .
      ?anat rdfs:label 'brain' .
      ?gene orth:organism ?o .
      ?o obo:RO_0002162 ?taxon .
      ?gene dcterms:identifier ?id .
      ?taxon up:commonName 'fruit fly' .
    }
  }
  SERVICE <https://sparql.omabrowser.org/lode/sparql> {
    ?cluster a orth:OrthologsCluster .
    ?cluster orth:hasHomologousMember ?node1 .
    ?cluster orth:hasHomologousMember ?node2 .
    ?node2 orth:hasHomologousMember* ?protein2 .
    ?node1 orth:hasHomologousMember* ?protein1 .
    ?protein1 dcterms:identifier ?id .
    ?protein2 rdfs:seeAlso ?OMA_LINK .
    FILTER ( ?node1 != ?node2 )
  }
}
```

Fig. 2. Example federated SPARQL query

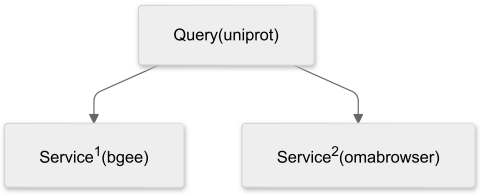

Fig. 3. Example service pattern tree

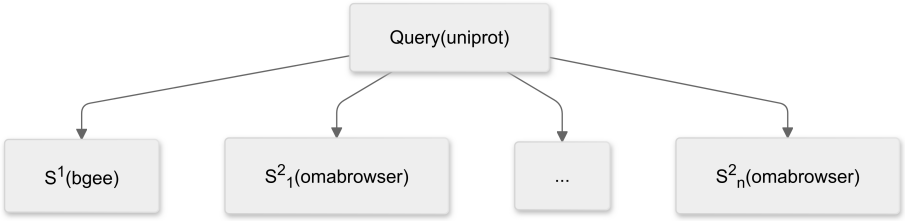

Fig. 4. Example service execution tree

The query (Fig. 2) is executed at the Oma-browser endpoint [2]. It is processed from top to bottom using the *nested loop join* execution strategy, where for each substitution of the SPARQL variable `?id` the second service pattern is executed. These executions are then aggregated into a new bulk execution node in the visualization.

Consider a scenario where the second service pattern subquery contains another nested service pattern. In such a case, it is impossible to substitute all service endpoints in the query with the proxy server endpoint at once when the query is submitted to the proxy server. Each proxy URL must encode the identifier of the parent in the service execution tree. However, for a nested service pattern, this identifier is only determined after a specific instance of

[2] https://sparql.omabrowser.org/lode/servlet/query

```
SELECT DISTINCT ?id ?OMA_LINK
WHERE
{
  SERVICE <sparql_debugger_proxy.org/service/query/78/parent/5676/serviceCall/0/endpoint/4> {
    SELECT DISTINCT  ?gene ?id {
      ?gene  a  <http://purl.org/net/orth#Gene> ;
                <http://purl.org/genex#isExpressedIn>  ?anat .
      ?anat   <http://www.w3.org/2000/01/rdf-schema#label>  "brain" .
      ?gene   <http://purl.org/net/orth#organism>  ?o .
      ?o      <http://purl.obolibrary.org/obo/RO_0002162>  ?taxon .
      ?gene   <http://purl.org/dc/terms/identifier>  ?id .
      ?taxon  <http://purl.uniprot.org/core/commonName>  "fruit fly"
    }
  }

  SERVICE <sparql_debugger_proxy.org/service/query/78/parent/5676/serviceCall/1/endpoint/6> {
    ?cluster a <http://purl.org/net/orth#OrthologsCluster> ;
        <http://purl.org/net/orth#hasHomologousMember>  ?node1 ;
        <http://purl.org/net/orth#hasHomologousMember>  ?node2 .
    ?node2 (<http://purl.org/net/orth#hasHomologousMember>)* ?protein2 .
    ?node1 (<http://purl.org/net/orth#hasHomologousMember>)* ?protein1 .
    ?protein1  <http://purl.org/dc/terms/identifier>  ?id .
    ?protein2  <http://www.w3.org/2000/01/rdf-schema#seeAlso> ?OMA_LINK
    FILTER ( ?node1 != ?node2 )
  }
}
```

Fig. 5. Request to a root query SPARQL endpoint generated by the debugging proxy server

```
SELECT  ?OMA_LINK ?protein1 ?protein2 ?node2 ?node1 ?cluster
WHERE
  { ?cluster a <http://purl.org/net/orth#OrthologsCluster> ;
       <http://purl.org/net/orth#hasHomologousMember>  ?node1 ;
       <http://purl.org/net/orth#hasHomologousMember>  ?node2 .
    ?node2 (<http://purl.org/net/orth#hasHomologousMember>)* ?protein2 .
    ?node1 (<http://purl.org/net/orth#hasHomologousMember>)* ?protein1 .
    ?protein1  <http://purl.org/dc/terms/identifier> "FBgn0000003".
    ?protein2  <http://www.w3.org/2000/01/rdf-schema#seeAlso>  ?OMA_LINK
    FILTER ( ?node1 != ?node2 )
  }
```

Fig. 6. Request to a SPARQL endpoint generated by the debugging proxy server

bulk service execution has started at the proxy server. As a result, only the service endpoint URLs at the first level of nesting in the query are initially replaced with proxy URLs. The remaining service endpoint URLs are gradually replaced as the nested service endpoints are invoked through the debugging proxy server, elevating them to the first level of nesting.

Each query can contain more than one directly nested service. In the service execution tree, these executions share the same parent, but it is necessary to distinguish which nested service they belong to in order to create the bulk execution node. This is achieved by encoding a sequential number for each nested service, designated as the `serviceCall` parameter, into the proxy URL during the proxy URL substitution process.

Following the sample BioSODA query, Fig. 5 presents an example of a request sent to the root query endpoint by the proxy server, while Fig. 6 presents a request sent to the second service. Both requests are generated by the proxy server, with endpoints being enumerated by it.

Consider that during bulk execution, the service execution tree may differ from the service pattern tree. This discrepancy can also occur when service endpoints apply optimizations, such as grouping triple patterns evaluated

by the same endpoint, as, for instance, with the *Exclusive Groups* in FedX [8]. Another such case is issuing special SPARQL ASK requests to determine data availability [7].

2.2. Performance issues

Queries can be traced by the debugging proxy server in parallel. Each execution of a service corresponds to an HTTP request handled by the proxy server. Service execution trees can become large and deeply nested. Moreover, some SPARQL engines can be capable of executing multiple service calls concurrently within the same query execution. Taken together, this creates significant performance pressure on the debugging proxy server, especially when handling parallel executions at scale.

Note that when a service is executed at an endpoint, it is initiated by a corresponding proxy service execution, which waits for the result. Additionally, all preceding nodes in the service execution tree must also wait for their corresponding service executions to complete. As a result, numerous parallel threads are required, many of which may be blocked while awaiting responses. To address this, Java Virtual Threads are utilized, allowing each proxy server request to be handled by its own virtual thread. The benefit of this approach is that virtual threads are lightweight, enabling the system to efficiently manage thousands of them. If virtual threads get blocked, they do not block system threads, ensuring that the system remains both responsive and scalable.

A feature allowing users to cancel queries, which terminates all virtual threads associated with a specific query execution, is also available. Additionally, this feature instructs the proxy server to reject any new proxy calls related to the query being cancelled, even if they are initiated by the original endpoints during service execution after the cancellation has begun.

2.3. Frontend

To visualize query execution tracing, an npm package that renders the service execution tree is provided. This package is implemented as an independent React component. The component's API consists of callbacks receiving a SPARQL query and the endpoint where it should be executed. These parameters are then sent to the proxy server, which initiates the query execution and starts notifying the component about updates to the service execution tree.

After query debugging is started, the service execution tree is rendered in real time. Instead of having the browser poll the server for updates, the proxy server actively pushes tree node changes to the browser, which re-renders only the affected parts of the tree dynamically. This real-time update is achieved using the Server-Sent Events (SSE) protocol [17]. Compared to the WebSocket protocol [19], SSE offers a simpler and more efficient solution for this use case, as it requires only one-way communication from the proxy server to the browser, following an initial handshake.

Note that it is impossible to determine when a bulk execution node has fully completed until the parent execution is finished. Therefore, the bulk execution time is only partially calculated during execution. Additionally, it is assumed that executions within the bulk can occur in parallel. As a result, the execution time is calculated as the time interval between the start of the first endpoint call and the completion of the last call within the bulk. Consequently, the displayed execution time continues to increase until it is definitively set when the parent execution node is completed.

Part of the visualized service execution tree, including bulk execution nodes, is shown in Fig. 7.

3. Discussion

The presented approach has been designed so that, during debugging, each query is evaluated in the same way as during direct execution. Nevertheless, the service execution trees may differ. For example, if a service endpoint is declared with the same URL as its parent, the service may bypass a nested service call, executing locally and potentially using internal optimizations. This can lead to discrepancies in the execution structure. In some cases, even the results can differ. Certain SPARQL endpoints, such as Wikidata, enforce a whitelist of allowed service URLs. If the proxy server is not included in this whitelist, service calls made during debugging may result in errors, while the same service call could succeed in a direct query execution without the proxy. This discrepancy occurs because the proxy is treated as an unauthorized endpoint during the debugging process.

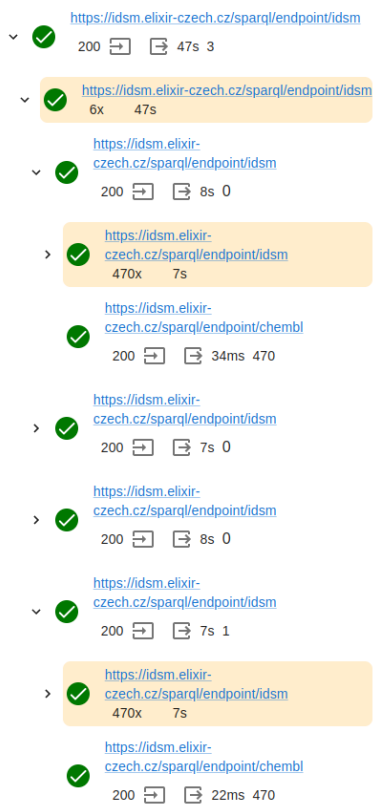

Fig. 7. Visualized service execution tree

3.1. Case study from practice

The usefulness of the tool in practice is demonstrated on the example of a query that returns no results, even though it is known that some solutions matching the query exist. The query (Fig. 8) should return *proteins that catalyse reactions involving cholesterol-like compounds*. It is evaluated by the Uniprot endpoint [10] and also includes nested services using the Rhea [1] and IDSM endpoints [3].

By tracing the query using the SPARQL debugger, it can be observed that the UniProt endpoint alters the original object term `"0.9"^^xsd:double` for the predicate `sachem:cutoff` to the term `0.9` in the innermost service call. According to the SPARQL specification, the IDSM endpoint interprets `0.9` as a decimal rather than a double. Because IDSM is strongly typed, this change results in IDSM not returning any results for the given subquery, which means that not even the entire query will return any results. As a workaround, user can replace `"0.9"^^xsd:double` in the query with `"9E-1"^^xsd:double`, which is interpreted as a double even if the type is stripped, resolves the issue.

In our experience, the Uniprot endpoint first tries to use the nested loop join strategy and execute the respective service for different substituted values. However, this approach takes a long time and does not lead to the desired outcome. In subsequent attempts, the endpoint typically choose the option to call the service directly without any substitution, which allows it to get the result in a short time.

These observations, made possible by the SPARQL debugger, demonstrate its practical utility.

```
SELECT ?CHEBI ?RHEA_REACTION ?PROTEIN WHERE {
  SERVICE <https://sparql.rhea-db.org/sparql> {
    SERVICE <https://idsm.elixir-czech.cz/sparql/endpoint/chebi> {
      # ChEBI compounds similar to cholesterol
      ?CHEBI sachem:similarCompoundSearch [
        sachem:query "C1C2(C3(CCC4(C(C3(CC=C2CC(C1)O))(CCC4(C(C)CCCC(C)C)))C))C";
        sachem:cutoff "0.9"^^xsd:double ]
    }

    # Rhea reactions involving compound ?CHEBI
    ?RHEA_REACTION rdfs:subClassOf rh:Reaction .
    ?RHEA_REACTION rh:status rh:Approved .
    ?RHEA_REACTION rh:side / rh:contains / rh:compound / rh:chebi ?CHEBI .
  }

  # Rhea reactions catalysed by UniProt proteins
  ?PROTEIN up:annotation/up:catalyticActivity/up:catalyzedReaction ?RHEA_REACTION.

  # UniProtKB/Swiss-Prot entries
  ?PROTEIN up:reviewed true.

  # Human entries
  ?PROTEIN up:organism taxon:9606.
}
```

Fig. 8. Retrieval of a list of UniProtKB/Swiss-Prot human proteins that catalyse Rhea reactions involving cholesterol-like compounds

4. Conclusion

The presented software provides SPARQL developers with a debugger tool designed to offer detailed insights into the execution of complex federated queries. It has already proven itself to be effective in practice and has helped identify and resolve several errors and performance issues that were previously impossible to simply address. The proxy server exposes a REST API that enables integration with other applications independently of the debugger frontend.

The source code for both the proxy server [3] and the frontend [4] is available on GitHub. Additionally, a deployment of the SPARQL federated query debugger[5] is ready for use online.

Acknowledgement

This work was supported by the CHIST-ERA grant TRIPLE, by the Technology Agency of the Czech Republic (TAČR) within the National Recovery Plan, project No. TH86010003. Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

References

[1] P. Bansal, A. Morgat, K.B. Axelsen, V. Muthukrishnan, E. Coudert, L. Aimo, N. Hyka-Nouspikel, E. Gasteiger, A. Kerhornou, T.B. Neto, M. Pozzato, M.C. Blatter, A. Ignatchenko, N. Redaschi and A. Bridge, Rhea, the reaction knowledgebase in 2022, *Nucleic Acids Res* **50**(D1) (2022), D693–D700. doi:10.1093/nar/gkab1016. https://www.ncbi.nlm.nih.gov/pubmed/34755880.

[3] https://github.com/iocbbioinf/sparql_debugger_server, https://doi.org/10.5281/zenodo.15282932
[4] https://github.com/iocbbioinf/sparql_debugger_component, https://doi.org/10.5281/zenodo.15295683
[5] https://sparql-debugger.elixir-czech.cz/

[2] C. Buil-Aranda, A. Polleres and J. Umbrich, Strategies for Executing Federated Queries in SPARQL1.1 (2014), 390–405. ISBN 978-3-319-11914-4. doi:10.1007/978-3-319-11915-1_25.
[3] J. Galgonek and J. Vondrasek, IDSM ChemWebRDF: SPARQLing small-molecule datasets, *J Cheminform* **13**(1) (2021), 38. doi:10.1186/s13321-021-00515-1. https://www.ncbi.nlm.nih.gov/pubmed/33980298.
[4] J. Pinero, J.M. Ramirez-Anguita, J. Sauch-Pitarch, F. Ronzano, E. Centeno, F. Sanz and L.I. Furlong, The DisGeNET knowledge platform for disease genomics: 2019 update, *Nucleic Acids Res* **48**(D1) (2020), D845–D855. doi:10.1093/nar/gkz1021. https://www.ncbi.nlm.nih.gov/pubmed/31680165.
[5] L. Rietveld and R. Hoekstra, YASGUI: Not just another SPARQL client?, *CEUR Workshop Proceedings* **1056** (2013), 1–9. ISBN 978-3-642-38708-1. doi:10.1007/978-3-642-41242-4_7.
[6] A. Rutz, M. Sorokina, J. Galgonek, D. Mietchen, E. Willighagen, A. Gaudry, J.G. Graham, R. Stephan, R. Page, J. Vondrasek, C. Steinbeck, G.F. Pauli, J.L. Wolfender, J. Bisson and P.M. Allard, The LOTUS initiative for open knowledge management in natural products research, *Elife* **11** (2022). doi:10.7554/eLife.70780. https://www.ncbi.nlm.nih.gov/pubmed/35616633.
[7] M. Saleem, A. Potocki, T. Soru, O. Hartig and A.-C. Ngonga Ngomo, CostFed: Cost-Based Query Optimization for SPARQL Endpoint Federation, *Procedia Computer Science* **137** (2018), 163–174. doi:10.1016/j.procs.2018.09.016.
[8] A. Schwarte, P. Haase, K. Hose, R. Schenkel and M. Schmidt, FedX: Optimization Techniques for Federated Query Processing on Linked Data (2011), 601–616. ISBN 978-3-642-25072-9. doi:10.1007/978-3-642-25073-6_38.
[9] SIB RDF Group Members, The SIB Swiss Institute of Bioinformatics Semantic Web of data, *Nucleic Acids Res* (2023).
[10] C. UniProt, UniProt: the universal protein knowledgebase in 2021, *Nucleic Acids Res* **49**(D1) (2021), D480–D489. doi:10.1093/nar/gkaa1100. https://www.ncbi.nlm.nih.gov/pubmed/33237286.
[11] M.D. Wilkinson, M. Dumontier, I.J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.W. Boiten, L.B. da Silva Santos, P.E. Bourne, J. Bouwman, A.J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C.T. Evelo, R. Finkers, A. Gonzalez-Beltran, A.J. Gray, P. Groth, C. Goble, J.S. Grethe, J. Heringa, P.A. t Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S.J. Lusher, M.E. Martone, A. Mons, A.L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M.A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao and B. Mons, The FAIR Guiding Principles for scientific data management and stewardship, *Sci Data* **3** (2016), 160018. doi:10.1038/sdata.2016.18. https://www.ncbi.nlm.nih.gov/pubmed/26978244.
[12] M. Zahn-Zabal, P.A. Michel, A. Gateau, F. Nikitin, M. Schaeffer, E. Audot, P. Gaudet, P.D. Duek, D. Teixeira, V. Rech de Laval, K. Samarasinghe, A. Bairoch and L. Lane, The neXtProt knowledgebase in 2020: data, tools and usability improvements, *Nucleic Acids Res* **48**(D1) (2020), D328–D334. doi:10.1093/nar/gkz995. https://www.ncbi.nlm.nih.gov/pubmed/31724716.
[13] Apache Jena: A Java Framework for Semantic Web and Linked Data Applications, Accessed: 2024-11-21. https://jena.apache.org/index.html.
[14] Datadog: Cloud Monitoring as a Service, 2024-11-19. https://www.datadoghq.com/.
[15] Exploring Biological Data Using SPARQL, Accessed: 2024-11-21. https://biosoda.expasy.org/build_biosodafrontend/.
[16] Query Federation with Comunica, Accessed: 2024-11-19. https://comunica.dev/docs/query/advanced/federation/.
[17] Server-sent events, Accessed: 2024-11-21. https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-event.
[18] Virtuoso Universal Server, 2024-11-21. https://virtuoso.openlinksw.com/.
[19] The WebSocket Protocol, Accessed: 2024-11-21. https://datatracker.ietf.org/doc/html/rfc6455.
[20] SPARQL 1.1 Federated Query, 2013, Accessed: 2024-11-19. https://www.w3.org/TR/sparql11-federated-query/.