Observations on automated client-side query federation over Wikidata SPARQL endpoints

Jonni Hanski, Elias Crum and Ruben Taelman

IDLab, Department of Electronics and Information Systems, Ghent University – imec

Abstract

The recent Wikidata graph split divided a previously singular SPARQL endpoint into two distinct ones, breaking existing queries that depend on the combined data from these endpoints. To accommodate this graph split, instructions for manual source assignment have been provided. However, the proposed solution of manual source annotations within the queries themselves, through the use of SPARQL `SERVICE` clauses, not only imposes additional work on users of these endpoints, but also assumes prior knowledge of which data come from which endpoint, and how they should be combined. Potential future graph splits would result in this manual source assignment having to be done again. Within this work, we employ client-side query federation over the two Wikidata endpoints, using state-of-the-art source assignment approaches for query operations, to demonstrate the feasibility and challenges of automated federation as an alternative to manual source assignment. Through our experiments, we show how client-side federation can offer a viable alternative to manual source assignment for certain queries, where the amount of data to process remains within client-side resource limits, and provided no custom behaviour is attached to standard SPARQL operations. Future work will be needed to address the trade-offs between network request counts and client-side data processing, to be able to execute queries that access large amounts of data from multiple sources.

Keywords

SPARQL, Wikidata, federated querying, VoID,

1. Introduction

The recent graph split by Wikidata[1] saw a previously unified SPARQL [1] endpoint [2] split in two. The split was done for scalability reasons, due to technical limitations imposed by the underlying software implementation. However, this also breaks queries that rely on the full dataset being available in the original endpoint. The endpoint, at present, offers no automated means of adjusting queries to account for this split, and the onus is thereby on the users composing and executing queries to ensure they extract the data from the endpoint where it exists, under pain of incomplete or inaccurate results.

The impact analysis by Wikimedia Foundation[2], taking into account a number of Wikidata-related tools alongside a representative sample of other queries from their logs[3], places the number of known affected queries at below 10%. Although this proves that the vast majority of queries continue executing successfully even after the graph split, it also means there is a non-negligible share of queries that fail, lest they be updated. To assist query authors with these updates, a set of documentation exists[4], accompanied by a federation guide[5] and a number of federated query examples[6]. These examples and documentation make use of the `SERVICE` clauses in the SPARQL query language, that allow forwarding parts of a query to another service endpoint. This requires manual modifications to the queries themselves, and the query author needs to be aware of the data distribution across both endpoints.

The 5th Wikidata Workshop for the scientific Wikidata community at the International Semantic Web Conference 2025, November 02–03, 2025, Nara, Japan

jonni.hanski@ugent.be (J. Hanski); elias.crum@ugent.be (E. Crum); ruben.taelman@ugent.be (R. Taelman)
0009-0004-0721-2169 (J. Hanski); 0009-0005-3991-754X (E. Crum); 0000-0001-5118-256X (R. Taelman)

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/WDQS_graph_split
[2]https://wikitech.wikimedia.org/wiki/Wikidata_Query_Service/WDQS_Graph_Split_Impact_Analysis
[3]https://gitlab.wikimedia.org/repos/search-platform/notebooks/-/tree/main/wdqs/T349512_representative_wikidata_query_samples
[4]https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/WDQS_graph_split
[5]https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/WDQS_graph_split/Internal_Federation_Guide
[6]https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/WDQS_graph_split/Federated_Queries_Examples

The alternative to manually assigning parts of a query to different sources is *virtual integration* [3, 4, 5, 6], where data from different sources is made available for querying as if it were centralised. This can be implemented through automated query federation over multiple data sources by the query engine itself [3], and the approaches to implementing it have been categorised as top-down, bottom-up, and mixed [7]. The top-down approaches rely on complete information at optimisation time, to produce a query plan based on an overview of the data source contents, and then executing it. Within distributed Linked Data contexts, this overview of data sources can be enabled through the use of an indexing system [8], with individual data source contents further summarised in their service descriptions [3]. The standard Vocabulary of Interlinked Datasets (VoID) [9] allows for the documentation of data source contents, such as information on predicate and class values, the number of triples with a given predicate value, or the number of subjects with a given class as their type. This information can also be used to estimate triple pattern cardinalities, using the set of formulae from Hagedorn et al. [10]. The bottom-up approaches, such as traversal-based query execution [11], perform source discovery and data retrieval during query execution, and are thereby limited in their advance optimisation capabilities, having to rely on heuristics. The mixed approaches rely on partial information to be present during query planning, and use corrective actions, such as join order adjustments, to perform additional optimisation when accurate metrics become available.

The focus of our work is the top-down approach, where the data sources – the two Wikidata SPARQL endpoints – are known beforehand. This allows us to use two prominent state-of-the-art approaches for source assignment to achieve client-side virtual integration: FedX and SPLENDID. The FedX [5] approach for source assignment uses ASK queries on all known sources, and does not rely on additional metadata being exposed by these data sources, making it the more universal solution. The SPLENDID [12] approach, on the other hand, makes use of VoID dataset descriptions, by first checking which data sources *may* contain data for given parts of a query, and then verifying these potential sources using SPARQL ASK queries as in FedX. The Wikidata SPARQL endpoints expose VoID dataset descriptions at their URIs, in addition to their SPARQL service descriptions. This allows for the use of the formulae from Hagedorn et al. [10] for cardinality estimation, as well as the use of these cardinality estimates for the initial source assignment in SPLENDID. Using either the FedX or SPLENDID approach, when the engine is able to identify *exclusive groups* [5] of operations only answered by a single source, these operations can be grouped together for more efficient execution at the specific source, reducing unnecessary data transfer and processing locally, as well as achieving source assignment for operations within a query that closely match manual assignment. When an operation can be answered by multiple sources, the data can be combined locally using a UNION over those sources, as with Semagrow [6].

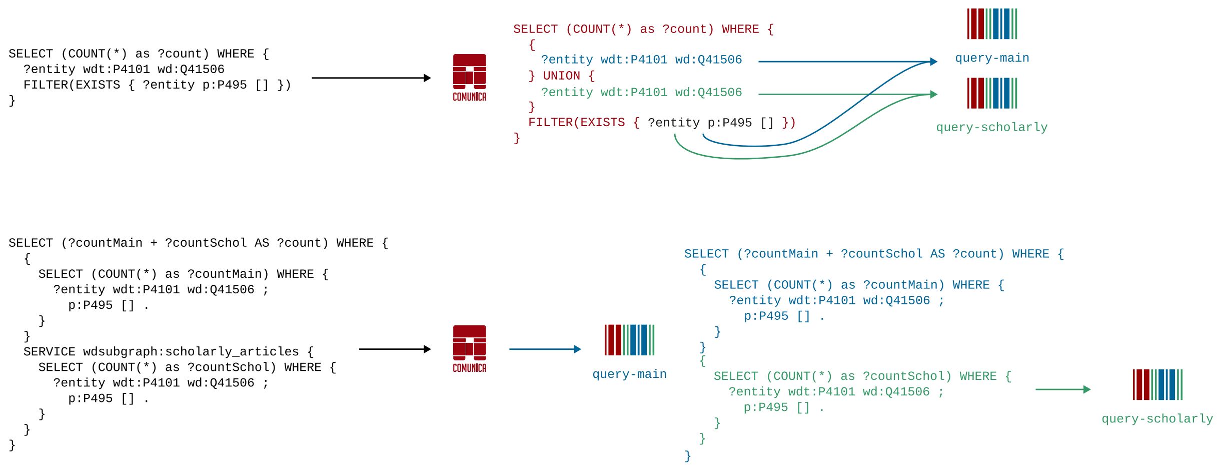
Within this work, we demonstrate the feasibility and challenges of client-side query federation over the Wikidata endpoints, using the FedX and SPLENDID approaches to source assignment. We compare the set of example federated Wikidata queries with manual server-side federation to their correspoding queries with automated client-side federation, and identify issues with client-side handling of large volumes of data, as well as with custom server-side handling of standard SPARQL query operations with specific values, that a client-side query engine could not be aware of, that can prevent successful source assignment and query execution in practice.

The remainder of this paper is structured as follows. Section 2 outlines the experiments performed within the context of this work, followed by Section 3 to analyse the results of these experiments, and Section 4 for the conclusions.

2. Experiments

The goal of our experiments is to explore the feasibility of automated client-side query federation, as an alternative to the manual server-side approach. Therefore, we performed our experiments with these two different means of dividing a query between endpoints:

1. The *automated* approach without SERVICE clauses, where the local query engine divides the operations between sources. The local query engine extracts the necessary information from



```
SELECT (COUNT(*) as ?count) WHERE {
  ?entity wdt:P4101 wd:Q41506
  FILTER(EXISTS { ?entity p:P495 [] })
}
```

```
SELECT (COUNT(*) as ?count) WHERE {
  {
    ?entity wdt:P4101 wd:Q41506
  } UNION {
    ?entity wdt:P4101 wd:Q41506
  }
  FILTER(EXISTS { ?entity p:P495 [] })
}
```

```
SELECT (?countMain + ?countSchol AS ?count) WHERE {
  {
    SELECT (COUNT(*) as ?countMain) WHERE {
      ?entity wdt:P4101 wd:Q41506 ;
        p:P495 [] .
    }
  }
  SERVICE wdsubgraph:scholarly_articles {
    SELECT (COUNT(*) as ?countSchol) WHERE {
      ?entity wdt:P4101 wd:Q41506 ;
        p:P495 [] .
    }
  }
}
```

```
SELECT (?countMain + ?countSchol AS ?count) WHERE {
  {
    SELECT (COUNT(*) as ?countMain) WHERE {
      ?entity wdt:P4101 wd:Q41506 ;
        p:P495 [] .
    }
  }
  {
    SELECT (COUNT(*) as ?countSchol) WHERE {
      ?entity wdt:P4101 wd:Q41506 ;
        p:P495 [] .
    }
  }
}
```

Figure 1: The *automated* approach (top) requires the local client-side query engine to split the operations between sources by itself, execute the partial queries at the two Wikidata endpoints, and perform the construction of the final results locally. The *manual* approach using service clauses (bottom) requires no processing from the local query engine, as it can simply forward the query to the main Wikidata endpoint as the only source, and pass through the results from this endpoint. The remote main endpoint is responsible for retrieving the additional data from the scholarly endpoint, as instructed by the service clause.

remote endpoints, passing data between them if necessary to properly handle intermediate bindings, and produces the final results locally. Within this approach, the query engine is responsible for appropriate source assignment of query operations.

2. The *manual* approach using SERVICE clauses, where the query author manually divides the operations between sources when composing the query. The local query engine fowards the full query to the primary remote endpoint, that forwards parts of the query to other sources in accordance with these clauses. Within this approach, the author of the query is responsible for appropriate source assignment of query operations.

The automated source assignment is implemented in three different ways, all assuming prior knowledge of the data source URIs, following the FedX [5] and SPLENDID [12] approaches to source assignment:

1. The ASK-based approach, as in FedX, where a query operation is sent to each source in an ASK query, to check whether the source contains any results for that operation.
2. The COUNT-based approach, functionally identical to ASK as in FedX, to account for scenarios where an endpoint does not support ASK queries.
3. The *VoID*-based approach, as in SPLENDID, where the initial source assignment is done based on VoID dataset descriptions, followed by ASK queries to eliminate false positive assignments.

The main difference between SERVICE-based federation and automated one is the distribution of responsibilities. When using SERVICE clauses, the endpoint receiving the full query is responsible for forwarding the contents of the clauses to their respective endpoints, and for constructing the final results for the query. Using the automated approach, the client assumes the responsibility of executing the query, and forwarding parts of it to the appropriate remote endpoints as needed. This is illustrated in Fig. 1. The system resources, such as network bandwidth or processor and memory allocation, may differ considerably between a remote endpoint and a local query engine. Therefore, the focus of our experiments is on the feasibility of client-side federation, and the practical challenges encountered, rather than absolute performance or resource consumption.

Within this work, we used the Comunica query engine framework[13], that allows client-side query federation over remote SPARQL endpoints. The engine allows fine-grained configuration of behaviour,

Query	Name in experiments	#TP	Property paths	Notable aggregates and other operations
1	author-birthday	2		limit
2	finding-duplicated-external-ids-with-a-group-by	1		group, order, having, count
3	joining-papers-and-authors	3		distinct, group, count, having
4	lookup-from-mwapi-results	3	alt	order, limit
5	number-of-articles-with-cito-annotated-citations-by-year	6	seq	group, order, count, distinct, min, if, select, optional
6	paper-subjects	2		limit
7	property-paths	4	*, +, alt, seq	order, distinct, select, sample, group
8	publications-wikiproject-author-not-in-project	3		distinct, minus
9	publications-wikiproject-main-subject-instance-of-person	3		distinct, limit
10	recent-publications	8	alt	group, order, min, sample, distinct, optional
11	simple-count	2		
12	simple-lookup-by-object-on-the-truthy-graph	2		order
13	simple-lookup-by-subject	2		
14	sitelinks-lookup	3		order

Table 1
Overview of the queries used for the experiments, extracted from the Wikidata federation tutorial and the sample federated queries. The number of triple patterns in *#TP* excludes the service parameters used by the endpoints. The nested `SELECT` clauses in query 11 were removed for the automated federation approach as unnecessary.

as well as component substitution, to help ensure fair comparisons between the algorithmic approaches, to avoid other engine implementation differences. The Comunica query engine also allows for the use of rate-limiting, that matches the client request rate to the server response rate. For example, if the server responds to one request every 100 milliseconds, then the query engine will send one request every 100 milliseconds. Following preliminary manual testing, we decided to enable this feature by default, due to the Wikidata SPARQL endpoints imposing a query budget of 60 seconds of processing within 60 seconds on each client[7]. Furthermore, individual queries are limited to 60 seconds in their duration. The query engine also supports the HTTP status code 429 and the `Retry-After` header, and we configured the engine to retry each failed request ten times, to ensure a request failed by rate-limiting would have a chance to be executed successfully. This rate-limiting approach, however, can not account for the total execution budget, and the engine supports no means of retrying queries that the remote endpoint terminates while the response is already being transferred to the client, so overlapping, long-running queries may still cause issues.

The queries used were taken from the Wikidata federation tutorial[8] and the example federated query set[9]. The Wikidata label helpers using `SERVICE` clauses with `wikibase:label` were replaced with `rdfs:label` extraction for all queries with a language tag filter, to ensure basic query functionality does not rely on Wikidata internal helpers. The queries vary in complexity, ranging from two triple patterns to eight of them, and from simple counts to complex subqueries with property paths and aggregates, as summarised in Table 1. Some of this complexity can be removed using client-side federation, such as most `SERVICE` clauses and their accompanying `UNION` operations. This also allows for the removal of some workarounds, that make use of a combination of `BOUND`, `IF`, and `COALESCE`, such as in query 5, which should also improve the readability of the queries and ease their composition.

The experiments were executed on the same retail commodity hardware machine, with an 8-core 16-thread CPU and 32 GB of memory, using *jbr.js*[10]. The query timeout was set to 5 minutes, following manual experiments where queries that consistently succeeded did so within this timeframe. The logical query plans were extracted from the engine separately from the query execution. These plans are the result of the query planning phase, with algebraic optimisation and source assignment complete, but before runtime optimisations by the engine, such as join entry ordering or physical operator selection for logical ones. This allows us to compare the source assignment between different automated approaches. Our experiments and their results are available online for transparency and reproducibility[11].

[7]https://www.mediawiki.org/wiki/Wikidata_Query_Service/User_Manual#Query_limits
[8]https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/WDQS_graph_split/Internal_Federation_Guide
[9]https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/WDQS_graph_split/Federated_Queries_Examples
[10]https://github.com/rubensworks/jbr.js
[11]https://github.com/surilindur/comunica-experiments/tree/main/experiments/wikidata-graph-split

Query	SERVICE + ASK			Automatic + ASK			Automatic + COUNT			Automatic + VoID + ASK		
	Duration (s)	Results	#HTTP	Duration (s)	Results	#HTTP	Duration (s)	Results	#HTTP	Duration (s)	Results	#HTTP
1	7.82	4	2	8.54	4	46	5.94	4	43	3.95	4	19
2	20.51	2,384	2	oom	-	8	oom	-	6	oom	-	6
3	2.76	0	2	mwbudget	-	149	mwbudget	-	144	32.09	0	215
4	2.97	10	2	mwapi	-	23	mwapi	-	17	mwapi	-	13
5	3.07	53	2	mwbudget	-	311	mwbudget	-	180	timeout	-	3,865
6	2.76	2	2	4.27	2	22	4.57	2	19	3.00	2	11
7	3.69	4	2	mwbudget	-	140	mwbudget	-	142	mwbudget	-	14
8	8.64	13	2	timeout	-	2,177	timeout	-	2,429	timeout	-	3,129
9	10.65	100	2	oom	-	266	mwbudget	-	222	mwbudget	-	608
10	2.92	4	2	mwbudget	-	68	mwbudget	-	93	mwbudget	-	162
11	2.66	1	2	timeout	-	2,154	timeout	-	2,548	146.22	1	1,325
12	45.28	1,815	2	mwbudget	-	10	mwbudget	-	73	mwbudget	-	1,281
13	2.70	3	2	5.39	6	30	116.69	6	56	4.02	6	22
14	3.09	140	2	122.31	140	1,138	120.10	140	1,132	71.30	140	572
Total			28			6,542			7,104			11,242
ASK			14			144			34			125
COUNT			-			4,785			5,384			-
Planning			50%			75%			76%			1%

Table 2
The SERVICE clause-based results from manual queries establish the ground truth and the baseline to compare against. The automated client-side federation approaches executed only 4 queries successfully, or 6 when using VoID descriptions to avoid COUNT queries, at the expense of considerably higher HTTP request counts. The queries that timed out locally are marked with *timeout*, the ones that ran out of memory with *oom*, the ones that failed after exceeding server-side execution budget with *mwbudget*, and the queries that failed due to the custom use of SERVICE with `wikibase:mwapi` with *mwapi*.

3. Results

The experiment results, summarised in Table 2, show the majority of queries failing when using client-side automated federation. This does not appear to be caused by source assignment itself, as the automated source assignment in the query plans match the manual assignment via SERVICE clauses, while operating under the assumption that no custom behaviour is attached to standard SPARQL query operations. Thus, the source assignment part of automated federation works, and the failures are the result of other factors.

The server-side manual federation approach resulted in a consistent HTTP request count of two requests per query: one query to establish the endpoint type, and another one to send the query. The automated client-side approaches produced much higher HTTP request counts, as expected, with the client-side engine being responsible for extracting data from the servers. The notable exception was query 2, that resulted in only 6-8 requests, due to the engine downloading excessive amounts of data from the servers and then crashing. This issue was caused by the engine having to evaluate the pattern `(?q, wdt:P244, ?extid)` at both sources, and process the aggregates on `?q` and `?extid` locally. The main endpoint contains 1.6 million matches for this pattern and the scholarly graph 14, and combined with the aggregates including GROUP BY, ORDER BY, HAVING, and COUNT, this becomes an overwhelming task for a client-side engine. When the query requires combining considerable amounts of data from both endpoints, the alternative to downloading all the data locally is to take intermediate results from one endpoint and pass them to the other one, such as with client-side nested loop joins or bind joins [14], at the cost of additional HTTP requests. This can be observed with query 11, for example, that contains only two triple patterns, and performs a count over their join result. One of the patterns is only matched by the scholarly endpoint, with 659 results, but the other pattern contains 71,031 results in the scholarly graph, and 2.6 million matches in the main graph. The engine therefore checks each binding from the scholarly graph at the main graph, and this can cause the engine to take too long to process the query. The same can be observed with query 8, where both endpoints contain results for the triple pattern `(?thesis, wdt:P5008, wd:Q111645234)` – 23,823 matches in the main graph, and 66,245 in the scholarly one – and the engine has to combine these with other data from the main endpoint to resolve the final query results. This takes too many HTTP requests, and causes the query to time out. The alternative of downloading the required data from the main endpoint would also fail, because the pattern `(?person, rdfs:label, ?personLabel)` has 631 million matches in the main endpoint, and thus the bindings for `?person` must all be checked at the endpoint, instead.

Beyond the number of HTTP requests, the type of queries being sent for query planning purposes also exhibits interesting trends. The ASK-based approach used around 75% of its HTTP requests on query planning, the COUNT-based approach 76%, and the VoID-based approach only 1%. Thus, through the use of VoID descriptions, the relative share of queries used to *execute* the query could be increased to 99% within these experiments, up from 24-25%. Furthermore, the VoID-based approach was able to avoid the Wikidata execution budget limit of 60 seconds every 60 seconds, that prohibited the other two approaches from finishing query 3, and allowed it to progress further with query 5, running into the local timeout, instead. The VoID-based approach was also able to skip all the COUNT queries to extract metadata during the execution of query 11, used by the engine to optimise join ordering at runtime, when the other approaches timed out due to sending COUNT queries for each binding they were checking at the endpoint. The Wikidata VoID description is therefore extremely useful for client-side federation over the endpoints.

Another noteworthy observation is the failure to execute query 4, that makes use of the MediaWiki API abstraction, implemented server-side. This abstraction is implemented through non-standard handling of `SERVICE` clauses, when the endpoint URI is `wikibase:mwapi`, and the contents of the clause are not processed as triple patterns, being converted into an external API call with specific parameters, instead. The client-side query engine does not implement the same abstraction, and instead treats the `wikibase:mwapi` URI as a query source, and fails to execute the query, due to there being no actual data source available at that URI. This demonstrates the unintended side-effect of custom server-side handling of specific values or query operations, that can inhibit the transfer of queries between engines, or between server and client-side execution models. The same caveat applies to the label service using `wikibase:label`, the use of which was removed from the queries for the experiments, but that would suffer from the same problem if left in.

4. Conclusions

Within this work, we demonstrated the feasibility of executing Wikidata query federations client-side, avoiding the use of manual source assignments via `SERVICE` clauses, as a means to ease the composition of queries taking advantage of data present in multiple SPARQL endpoints. This *virtual integration* allows for easier use of the data, by reducing the amount of knowledge needed of its distribution, and by reducing the complexity of the queries themselves through the removal of `SERVICE` clauses, unions, and the associated bindings workarounds.

Through our experiments, we identified three major issues with our client-side federation approach: the data volume, the server-side execution budget, and the use of custom behaviour for standard query operations. Attempting to combine large volumes of data from two different endpoints may result in the client-side engine exceeding its resource allocations, such as running out of memory. Although this can be addressed through the use of bind join or nested loop join, to pass intermediate results between endpoints to avoid downloading all of the original data locally, the overhead of additional HTTP requests from applying these join algorithms can in turn cause queries to time out, or risk running into server-side request rate limits, the latter of which imposes the greater challenge for future work. The Wikidata SPARQL endpoints perform rate-limiting using a query execution budget, which can not be addressed by simple request rate limiters, and would require the client-side query engine to track the available budget client-side, and split the queries into manageable chunks to be sent to the server. For example, a single query chunk should execute within 60 seconds, but the engine should also avoid sending too many short queries per minute, or else the total execution time limit of 60 seconds every 60 seconds would be reached. Although the balancing of trade-offs between server-side and client-side execution have previously been investigated in the context of Triple Pattern Fragments (TPF) [15], such work introduced an entirely new interface type on an axis between data dumps and SPARQL endpoints, and perhaps such fragmentation could be avoided through the use of additional federation algorithms. Additionally, any server-side custom handling of standard SPARQL operations would need to be replicated client-side for all queries to execute.

Despite these challenges, there were several queries that successfully finished with correct results using client-side federation. This demonstrates the feasibility of client-side execution for some queries, and motivates future research into the direction of budget-aware rate-limiting and query partitioning, to help ensure clients adhere to server-side limits beyond simple request rates. Additional techniques should also be investigated to assist client-side engines in combining large volumes of data from multiple endpoints, without running into system resource limitations or being bottlenecked by excessive network requests.

Acknowledgments

This work was supported by SolidLab Vlaanderen (Flemish Government, EWI and RRF project VV023/10), as well as the CHIST-ERA grant TRIPLE (CHIST-ERA-22-ORD-09). Elias Crum is a PhD fellow of the Research Foundation – Flanders (FWO) (1S27825N). Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1202124N).

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] S. Harris, A. Seaborne, SPARQL 1.1 Query Language, W3C Recommendation, W3C, 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[2] L. Feigenbaum, G. T. Williams, K. G. Clark, E. Torres, SPARQL 1.1 Protocol, W3C Recommendation, W3C, 2013. https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/.

[3] B. Quilitz, U. Leser, Querying distributed RDF data sources with SPARQL, in: Proceedings of the 5th European Semantic Web Conference, 2008, pp. 524–538.

[4] O. Hartig, C. Bizer, J.-C. Freytag, Executing SPARQL Queries over the Web of Linked Data, in: Proceedings of the 8th International Semantic Web Conference, 2009, pp. 293–309.

[5] A. Schwarte, P. Haase, K. Hose, R. Schenkel, M. Schmidt, FedX: Optimization Techniques for Federated Query Processing on Linked Data, in: Proceedings of the 10th International Semantic Web Conference, 2011, pp. 601–616.

[6] A. Charalambidis, A. Troumpoukis, S. Konstantopoulos, SemaGrow: Optimizing Federated SPARQL queries, in: Proceedings of the 11th International Conference on Semantic Systems, 2015, pp. 121–128.

[7] G. Ladwig, T. Tran, Linked Data Query Processing Strategies, in: Proceedings of the 9th International Semantic Web Conference, 2010, pp. 453–469.

[8] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, J. Umbrich, Data Summaries for On-Demand Queries over Linked Data, in: Proceedings of the 19th International Conference on World Wide Web, 2010, pp. 411–420.

[9] R. Cyganiak, K. Alexander, J. Zhao, M. Hausenblas, Describing Linked Datasets with the VoID Vocabulary, W3C Note, W3C, 2011. https://www.w3.org/TR/2011/NOTE-void-20110303/.

[10] S. Hagedorn, K. Hose, K.-U. Sattler, J. Umbrich, Resource Planning for SPARQL Query Execution on Data Sharing Platforms, in: Proceedings of the 5th International Conference on Consuming Linked Data, 2014, pp. 49–60.

[11] O. Hartig, M. T. Özsu, Walking without a Map: Ranking-Based Traversal for Querying Linked Data, in: Proceedings of the 15th International Semantic Web Conference, 2016, pp. 305–324.

[12] O. Görlitz, S. Staab, SPLENDID: SPARQL Endpoint Federation Exploiting VoID Descriptions, in: Proceedings of the 2nd International Conference on Consuming Linked Data, 2011, pp. 13–24.

[13] R. Taelman, J. Van Herwegen, M. Vander Sande, R. Verborgh, Comunica: A Modular SPARQL Query Engine for the Web, in: Proceedings of the 17th International Semantic Web Conference, 2018, pp. 239–255.

[14] L. M. Haas, D. Kossmann, E. L. Wimmers, J. Yang, Optimizing Queries across Diverse Data Sources (1997) 276–285.

[15] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, P. Colpaert, Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web, Journal of Web Semantics 37 (2016) 184–206.