SHARQL: Shape Analysis of Recursive SPARQL Queries

Angela Bonifati, Wim Martens, Thomas Timm

SHARQL: Shape Analysis of Recursive SPARQL Queries

Angela Bonifati
Lyon 1 University

Wim Martens
University of Bayreuth

Thomas Timm
University of Bayreuth

ABSTRACT

We showcase SHARQL, a system that allows to navigate SPARQL query logs, can inspect complex queries by visualizing their shape, and can serve as a back-end to flexibly produce statistics about the logs. Even though SPARQL query logs are increasingly available and have become public recently, their navigation and analysis is hampered by the lack of appropriate tools. SPARQL queries are sometimes hard to understand and their inherent properties, such as their shape, their hypertree properties, and their property paths are even more difficult to be identified and properly rendered. In SHARQL, we show how the analysis and exploration of several hundred million queries is possible. We offer edge rendering which works with complex hyperedges, regular edges, and property paths of SPARQL queries. The underlying database stores more than one hundred attributes per query and is therefore extremely flexible for exploring the query logs and as a back-end to compute and display analytical properties of the entire logs or parts thereof.

KEYWORDS

SPARQL, database queries, query logs

1 INTRODUCTION

As large SPARQL query logs are being disclosed to the community, more and more queries are becoming available for advanced analytics [2, 4, 7]. Examples are the recently publicly available Wikidata query logs and the DBpedia logs which can be obtained from OpenLink Software. These query logs are massive and highly informative as they allow to increase the knowledge that we have about real-world queries. The latter understanding was poorly acquired for relational queries, often unavailable due to privacy concerns of their respective business applications. In contrast, Web-based query endpoints are making this knowledge acquisition possible and desirable for guiding the research community in many directions, such as graph query evaluation and optimization, graph query language design, and graph database tuning. In this demo, we focus on the presentation of massive SPARQL query logs, amounting to a corpus of roughly 0.8B queries from disparate data sources (including DBpedia and Wikidata as the most significant subsets of the corpus), as result of our previous work in the area [2, 4]. SHARQL builds on DARQL [3], a system deployed on top of a relational DBMS (PostgreSQL 10) with a web-based front end alongside a batch processing system (for loading and analyzing query logs, writing to files) and initially conceived for DBpedia queries. DARQL [2, 3] was however working on an initial batch of DBpedia logs augmented with logs from other sources and roughly amounting to 1/5 of our current huge corpus in SHARQL. Due to the fact that additional large query logs have been added since then, the new system SHARQL has a swathe of new features including:

- Adaptive edge rendering suitable for hypergraph visualization of queries; coupled with tree decomposition visualization.
- Precise shape analysis (with/out constants) and property paths support in visualization and shape analysis.
- A much larger query fragment wrt shape analysis covering C2RPQs (Conjunctive Two-ways Regular Path Queries), which is 57x times larger than the fragment of recursive queries found in previous logs (including DBpedia).
- Precise analytics on a rich variety of query features, such as spanning size, diameter, cycle lengths of queries alongside the numbers of nodes with branching, maximum node degree, edge cover and total sum of triples and symbols in

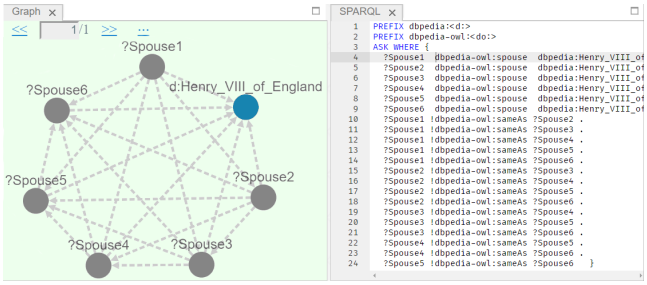


Figure 1: The Henry VIIIth Query

property paths. These analytics can be obtained for each particular class of queries exhibiting one of the discovered query shapes as well as for organic ("human-written") and robotic ("automatically generated") queries and for timeout and well-executed queries.

The complete code base of SHARQL is available on Github (https://github.com/PoDMr/sharql). To the best of our knowledge, graph query rendering has received fairly less attention than graph instance visualization, yet being relevant to open challenges of Visualization/HCI communities (e.g. progressive queries [5]).

2 DEMO OVERVIEW

SHARQL allows to visualize the queries of SPARQL log files with varying characteristics (Wikidata, DBpedia, BioPortal, LGD, OpenBioMed, Semantic Web Dog Food, British Museum), out of which there are recently publicly available logs [7]. Furthermore, the system is able to digest a huge amount of queries, currently about 800M in total. To further stimulate the interaction at the demo session, we prepare the following scenarios.

2.1 Rendering and Visualization of Queries

In order for a human to quickly understand queries, visualization may be very helpful. For instance, consider the query in Figure 1, coming from DBpedia. The query consists of 21 edges involving one constant ("Henry VIII") and six variables. Whereas the query itself (Fig 1 right) takes some time to parse, a visualization immediately shows that its shape is a 7-clique and that the user may be interested in obtaining the six spouses of Henry VIII.

Furthermore, the graph- or hypergraph structure of queries gives crucial information concerning the complexity of evaluating them. In the case of *conjunctive queries*, it is well known that acyclic queries can be evaluated efficiently, whereas queries that are highly cyclic are very complex. (For instance, if a query's shape is a k-clique, then evaluating it is equivalent to solving the NP-complete k-clique problem.)

Central aspects to query visualization are the rendering of *constants* and *variables*, *property paths*, and *hypergraph representations*. We will prepare a set of queries to show visitors how these aspects are visualized.

Hypergraph rendering of queries and navigation. In the visualization of SPARQL queries, the necessity for hyperedges (edges with more than two nodes) arises when rendering queries that use complex subqueries using the VALUES, SERVICE, BIND and FILTER operators from SPARQL. Hyperedges may also be necessary when variables are used in the *predicate* position of graph pattern triples, but we noticed that in practice this is needed much less often than with the aforementioned keywords.

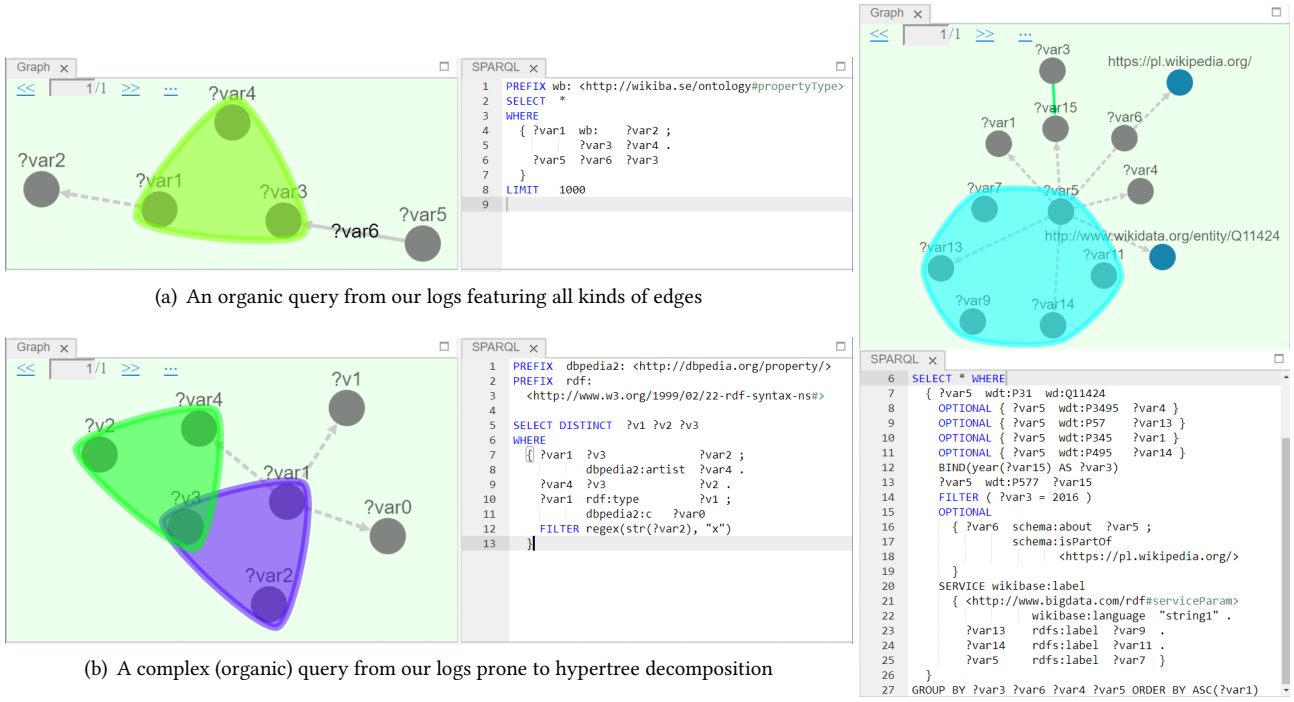
We therefore implemented a hypergraph rendering algorithm, which we will show in action here. Figure 2 contains three example queries and their rendering as hypergraph. We let the user navigate through our massive logs and visualize the hypergraph rendering of queries, starting with a few selected queries like the ones in Figure 2. Query log exploration can then be done in many different directions: we ran about 120 tests on every query in the logs. Any of these tests can be used for further exploration. For instance, users can search for queries with the largest hyperedge in the logs; focus on queries within the subsets of bot queries or user queries; focus on timeout queries, etc. In Section 3, we explain our novel hypergraph rendering of queries and the rendering of Fig. 2 in detail. We note that all our graph layout algorithms are interactive: the user can click a node and drag it, and the layout changes dynamically.

Property path visualization and related types. Property paths are extremely common in Wikidata query logs [4]. However, they can consist of complex expressions and can therefore quickly overcrowd the visualization. We solved this by rendering property paths as special (dashed) edges, without any annotation, for which the property path expression can be seen by hovering over the edge.

Impact of constants on query shapes. We want to show how the shape of queries changes if constants are removed from their (hyper-)graph. We noticed that the shape of many queries becomes *disconnected* if this happens [4]. This means that we can show on actual examples that it makes much sense for query evaluation to start with searching the constants in the query in the data and expand to variables later. This avoids the computation of large joins that are only pruned later when the constants are added.

2.2 Flexible Analytics

We have subjected each query in our database to a total of roughly 120 tests, compared to 62 we did in DARQL [3]. These tests involve information on the set of operators it uses, its number of triples, its graph shape, its hypergraph shape,

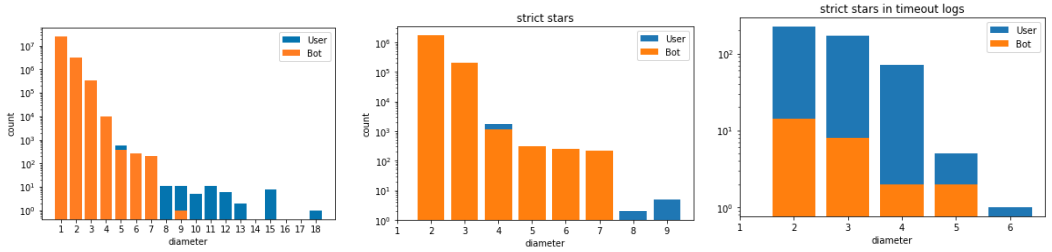


(a) An organic query from our logs featuring all kinds of edges

(b) A complex (organic) query from our logs prone to hypertree decomposition

(c) A complex organic query from our logs

Figure 2: Partial screenshots of our query viewer and visualizer for example queries.



(a) Diameter distribution (user vs. bot)

(b) Diameter distribution (user vs. bot) for star queries only

(c) Diameter distribution (user vs. bot) for star timeout queries only

Figure 3: Examples of diameter distributions on the entire Wikidata logs and subsets thereof

well-designedness, weak well-designedness, acyclicity, free-connex acyclicity, whether it uses property paths, number of variables, number of constants, etc.

For queries that can be rendered as a graph, we test if it has self-loops (self-joins), parallel edges, we measure the diameter of its graph (longest distance between nodes), maximum degree, its number of nodes with high degree, length of its shortest and longest simple cycles. Furthermore, our database records, if available, whether the query was *robotic* or *organic* and whether it timed out or not. SHARQL allows to use each of these analysis results to study subgroups of queries in detail. Under the hood, each query is stored together with 120 attributes in our query database, each of which represents the outcome of one of our tests.

```
qu3 = (
    'SELECT "depth_max", count(id) FROM "Queries"',
    "WHERE \"originMajor\" = 'wd_user_500'",
    ' AND star = true AND chain = false ',
    'GROUP BY "depth_max"'
)
qb3 = (
    'SELECT "depth_max", count(id) FROM "Queries"',
    "WHERE \"originMajor\" = 'wd_bot_500'",
    ' AND star = true AND chain = false ',
    'GROUP BY "depth_max"'
)
ru3 = query(qu3)
rb3 = query(qb3)
```

```
barchart(ru3, rb3, 'strict stars in timeout logs')
```

Figure 4: Jupyter code we used to produce Fig 3(c)

Analytics on the entire logs and subsets thereof. We can use our database as a back-end for Jupyter, which means

that, for each of the individual analyses we did, bar charts or other charts (scatter plots, etc.) can readily be produced. Already these simple distributions show interesting insights on global properties of the logs.

But the integration with Jupyter is much more flexible and we can combine statistics in an almost arbitrary manner. We discuss one scenario as an example. Figure 3(a) shows a bar chart, obtained using SHARQL, with the distribution of the diameter of the graphs of user versus bot queries in Wikidata, rougly 208M queries. (Notice the log scale on the vertical axis.) A user may be interested in refining this result to, say, star-shaped queries only, for which we see the result in Figure 3(b). (Here, "star" is a predefined shape we identified in [4] and which is very prominent in the logs.) The statistic can be refined even more by, e.g., only focusing on *timeout* star queries, see Figure 3(c). We can observe from the three views that user queries have significantly larger diameters than their robotic counterparts (Figure 3(a)). This is confirmed by star queries (Figure 3(b)) whereas for timeout queries the percentages are more balanced (Figure 3(c)). As an example, we give the Jupyter code we used to produce Figure 3(c) in Figure 4. When we computed these results, the system quite responsive and answered (produced the output bar charts) within a matter of seconds. The combination of the SHARQL database and Jupyter is extremely flexible and gives the user more than 0.8B queries to analyze.

Search for Complex Queries. Besides searching for simple queries in our large corpus, the tool allows us to quickly search for queries by size (so the largest ones can be found quickly), with complex structures (e.g., cyclic queries) and with advanced keywords. Since our corpus encompasses a varied set of log files coming from disparate SPARQL endpoints, SHARQL lets the users access the lineage of the queries under inspection and have a perception of what logs contain queries with certain complex characteristics.

Hypertree decomposition and its visualization. A *hypertree decomposition* of a hypergraph (similar to a tree decomposition of a graph) is a suitable clustering of its hyperedges yielding a tree or a forest. Such decompositions are important for database queries, since they can serve as a guide for join orderings. In SHARQL we can derive the hypertree decomposition of all queries in the logs that are sufficiently close to a *conjunctive query* so that hypertree decompositions make sense, as discussed in detail in [4]. The queries eligible for these decompositions amount to roughly 70% of the entire dataset.

3 GRAPH RENDERING

In this section, we explain some visualization aspects in more detail. Our query visualizer can render the edges of a query as regular edges or hyperedges. Regular edges have two nodes (and possibly an edge label) and are either dashed or solid. A dashed regular edge is an edge for which the edge label is hidden, and for which the label appears when hovering over it. This is especially helpful for property paths, since these expressions are arbitrarily complex. Solid regular edges may be grey or colored. If it is colored (as the green edge in Fig. 2(b)), it means that it is generated from a SERVICE, BIND, VALUES, or FILTER clause. (The green edge in Fig. 2(c) is generated from the BIND clause of the query.)

Hyperedges are edges that have more than two nodes. They can be generated in two scenarios. The first scenario occurs with a SERVICE, BIND, VALUES, or FILTER clause, which creates a constraint (and thus a hyperedge) between three or more variables or constants, as illustrated in Figures 2(c) for the blue edge and 2(b) for the green and purple edges. The second scenario is illustrated in Figure 2(a). Here, the green hyperedge arises from an ordinary triple pattern $\langle s\ p\ o \rangle$, for which the p position, ?var3, is a variable that is used elsewhere. Notice that we can deal with the triple pattern ⟨?var5 ?var6 ?var3⟩ differently. Since ?var6 is not used elsewhere, we can render it as a regular labeled edge.

As shown in the screenshots, we also render variables and constants in a different color. This is crucial for being able to judge the complexity of evaluating a query, because a constant can only match to one node in the graph, whereas a variable can potentially match to any node.

In drawing queries as hypergraphs, we have used the notion of planarity introduced in [6]. Not all hypergraphs have a vertex-planar representation, but in the majority of the cases in our queries this is the case. (In general, determining if a given hypergraph has a vertex-planar representation is NP-complete [6].) We adapted the layout algorithm in [1] in order to handle the visualization of ordinary edges and hyperedges in our corpus.

Acknowledgement. This work was supported by grant MA 4938/4-1 from the Deutsche Forschungsgemeinschaft (DFG).

REFERENCES

[1] N. A. Arafat and S. Bressan. Hypergraph drawing by force-directed placement. In *DEXA*, p. 387–394, 2017.
[2] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *PVLDB*, 11(2):149–161, 2017.
[3] A. Bonifati, W. Martens, and T. Timm. DARQL: deep analysis of SPARQL queries. In *The Web Conference (WWW)*, p. 187–190, 2018.
[4] A. Bonifati, W. Martens, and T. Timm. Navigating the Maze of Wikidata Query Logs. In *The Web Conference (WWW)*, p. 127–138, 2019.
[5] J.-D. Fekete and D. Fisher and A. Nandi and M. Sedlmair. Progressive Data Analysis and Visualization (Dagstuhl Seminar 18411). In *Dagstuhl Reports 8(10)*, p. 1–40, 2018.
[6] D. S. Johnson and H. O. Pollak. Hypergraph planarity and the complexity of drawing venn diagrams. *Journal of Graph Theory*, 11(3):309–325, 1987.
[7] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In *ISWC*, p. 376–394, 2018.