DARQL: Deep Analysis of SPARQL Queries[*]

Angela Bonifati
Lyon 1 University
angela.bonifati@univ-lyon1.fr

Wim Martens
University of Bayreuth
wim.martens@uni-bayreuth.de

Thomas Timm
University of Bayreuth
thomas.timm@uni-bayreuth.de

ABSTRACT

In this demonstration, we showcase DARQL, the first tool for deep, large-scale analysis of SPARQL queries. We have harvested a large corpus of query logs with different lineage and sizes, from DBPedia to BioPortal and Wikidata, whose total number of queries amounts to 180M. We ran a wide range of analyses on the corpus, spanning from simple tasks (keyword counts, triple counts, operator distributions), moderately deep tasks (projection test, query classification), and deep analysis (shape analysis, well-designedness, weakly well-designedness, hypertreewidth, and fractional edge cover). The key goal of our demonstration is to let the users dive into the SPARQL query logs of our corpus and let them discover the inherent characteristics of the queries.

The entire corpus of SPARQL queries is stored in a DBMS. The tool has a GUI that allows users to ask sophisticated analytical queries on the SPARQL logs. These analytical queries can both be directly written in SQL or composed by a visual query builder tool. The results of the analytical queries are represented both textually (as SPARQL queries) and visually. The DBMS performs the searches within the corpus quite efficiently. To the best of our knowledge, this is the first demonstration of this kind on such a large corpus and with such a number of varied tests.

CCS CONCEPTS

• **Information systems → Query languages for non-relational engines**; **Query log analysis**; • **Theory of computation** → *Database query languages (principles)*;

KEYWORDS

RDF, SPARQL, Conjunctive Queries, Query Analysis

1 INTRODUCTION

A plethora of SPARQL endpoints[1] is proliferating on the Internet thus allowing ordinary users to specify their queries either via APIs or manually. This phenomenon is leading to a democratization of query formulation. The queries are collected into log files by their respective owners and represent a valuable resource for understanding users' preferences and needs in terms of query specification, but also for guiding us in research on query language design, query evaluation and benchmarking [1, 2].

Motivated by previous studies on SPARQL log analysis [3, 4], we could harvest a large and varied corpus of SPARQL query logs amounting to a total of 180M queries [1], which is several orders of magnitude more than earlier analytical studies on SPARQL query logs. In this demonstration, we showcase DARQL, a tool for deep and fast analysis of large SPARQL query logs. The tool comes equipped with an extensive set of pre-defined tests, including simple tasks (keyword counts, triple counts, operator distributions), moderately deep tasks (projection test, query classification), and deep analysis (shape analysis, well-designedness, weakly well-designedness, hypertreewidth, and fractional edge cover). The primary goal of our demonstration is to let the users dive into the SPARQL query logs of our corpus and let them discover the inherent characteristics of the queries. A secondary goal is to advertise DARQL as an easy-to-use tool for SPARQL query analysis in the research community. Out of the box, DARQL analyzes 62 properties per query. We believe that DARQL will give researchers who want to dive into query log analysis a significant head start. Indeed, in our former analytical study [1] we only scratched the surface when it comes to finding correlations between query properties.

We will demonstrate our tool by using queries from the same corpus considered in [1]. Since many of these queries are not publicly available, the demonstration may give visitors a unique opportunity to get an idea of how queries actually look like in practice. We release the tool itself at https://github.com/PoDMr/darql.

2 SYSTEM AND MAIN COMPONENTS

A query builder lets the user modify the features of the queries under scrutiny to respectively enlarge or restrict the scope of the analyzed portion of the corpus. Since our tool is deployed on top of a relational DBMS with a web-based front end, each search on the corpus corresponds to an SQL query issued on the database. The user can also manually modify this query and rerun a deeper or coarser analysis at will.

Internally, the system consists of the following main components:

- a batch processing system (for loading and analyzing query logs, writing to files),
- a PostgreSQL 10 database, and
- a GUI served from a connecting back-end.

We refer to Figure 1 for a slightly more detailed overview. We describe the main components next.

2.1 Loading and Analyzing Query Logs

We release DARQL as open source project and therefore also discuss some aspects that will not be part of the demo, such as the batch processing system. Before the user can start exploring queries through the database interface, the queries need to be analyzed, deduplicated, and stored into the system. To this end, we provide scripts

[*]This work is supported by the Deutsche Forschungsgemeinschaft under Grant No: MA 4938/2–1.

[1]https://www.w3.org/wiki/SparqlEndpoints

WWW '18 Companion, April 23–27, 2018, Lyon, France
© 2018 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC BY 4.0 License.
ACM ISBN 978-1-4503-5640-4/18/04.
https://doi.org/10.1145/3184558.3186975

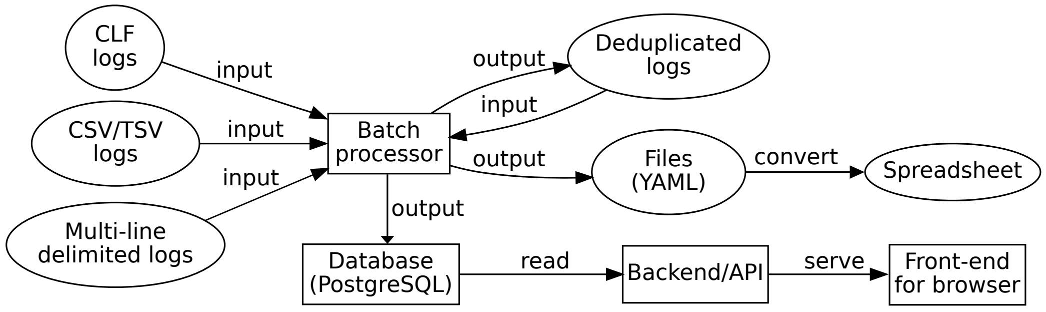


Figure 1: General architecture of the DARQL system.

that can handle three main log formats: CLF-based logs (Common Log Format) used by most web servers, delimiter-based line log formats (e.g. CSV, TSV), or multi-line delimited formats. Every query in the log is parsed and stored into the database. For queries that do not parse, we record this in the database and do not perform further analysis. Every query that parses is run through an extensive set of analytical tests: 1 parse test, 31 keyword tests (query type, operators, solution modifiers, aggregation operators, . . .), 8 simple structural tests (property path, projection, . . .), 4 well-designedness tests, 3 classifications into different kinds of conjunctive queries, 11 complex shape tests for the structure of conjunctive queries (chain, star, cycle, tree, flower,. . .), and 4 value tests (number of triples of the queries, hypertreewidth, fractional edge cover, and the origin of the query logs). These tests include (but are not limited to) all those that have been used in [1].

Prior to analysis, we test for duplicates in the query logs. We use SHA-256 for hashing to detect potential duplicates. The strings of queries are normalized by outputting them with the Jena parser, which normalizes whitespace and formats the queries in a readable form for our GUI. For some of the logs, we also need to add implicit prefixes explicitly to make queries valid as standalone queries. If we discover duplicate queries we do not re-run query analysis, but simply record its occurrence by referring to the first occurrence and store the new origin (log file and line number). As such, our system can display, for each query, how many duplicates were found and where.

The batch processing system can write output to either files or databases. It also allows logs to be rewritten in different formats, and deduplicated in a normalized form. The analysis output can be transformed to spreadsheets.

2.2 Database

The database is a PostgreSQL 10 system that stores the results of each analysis for every query. For duplicate log entries it stores the origin of each entry. We tested the database with the dataset of roughly 180M SPARQL queries from [1]. This dataset is about two orders of magnitude larger than those found in earlier studies. We used PostgreSQL 10 in order to utilize new parallelization features for queries and joins, which after appropriate tuning resulted in much faster query times compared to PostgreSQL 9.6.

We tested the system through a Web interface on our server[2] and noticed that most queries that our Query Builder generates are done in less than a second (e.g. 200ms).[3] Count queries are generally more expensive in PostgreSQL and take between 2–4 seconds on large sets (50–100 million queries) and are faster if subsets are smaller. In order to achieve this performance, we created a set of indexes. Index creation can take up to 2–4 minutes.

All non-duplicate queries are stored in a single table `Queries` with the information of analysis results as boolean or numeric columns. Each query is assigned a unique id. Additionally, it contains the string of the query, a hash of this string, and its origin (data set, filename, line number in file). Duplicates are stored in a table `Duplicates`, containing an id for the duplicate, a reference id to the original query, and the origin of the duplicate.

2.3 User Interface

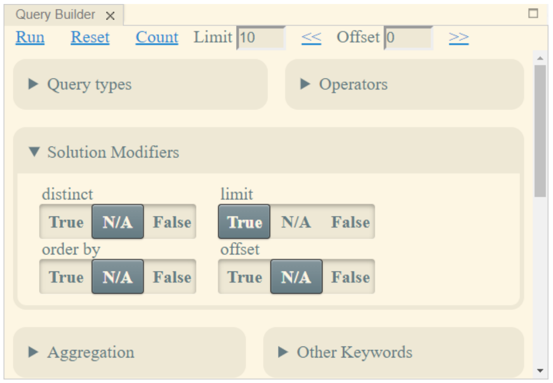
The user interface consists of several components that are connected. The main ones are: (1) a query builder, (2) an SQL text editor, (3) a query visualizer, (4) a SPARQL text display, and (5) a query result table display.

In a typical usage scenario of the tool, the user employs the query builder to select properties that she is interested in. Figure 2(a) shows a partial screenshot of the query builder. Under "Query Types", one can click if one wants to search for SELECT, ASK, CONSTRUCT, or DESCRIBE queries. Likewise, we have categories for "Operators", "Solution Modifiers", "Well Designedness", "Shapes", etc. As Figure 2(a) shows, groups of properties can be folded and unfolded at need. Boolean properties can be set to True, False, or N/A (don't care). For numerical properties (e.g., number of triples, hypertreewidth), we simply allow that the user enters the desired number or range.

When the user clicks "Run", the GUI automatically generates an SQL query that fetches up to ten queries from our query logs and displays it in the SQL text editor, see Figure 2(b) for an example query. The SQL query in the editor is fully editable, in case the user wants to refine the search.

Upon executing the SQL query, several things happen at once:

[2] A 2-CPU Intel Xeon E5-2630v2 2.6 GHz server with 128GB RAM and running Ubuntu 16.04 LTS.
[3] We discuss the Query Builder in more detail in Section 2.3.


Query Builder

Run Reset Count Limit 10 << Offset 0 >>

▶ Query types ▶ Operators

▼ Solution Modifiers

distinct: True [✓ N/A] False

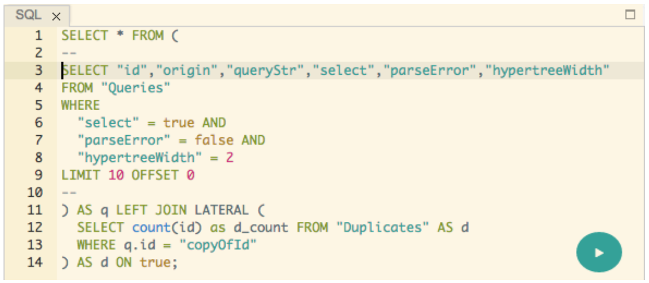
limit: [✓ True] N/A False

order by: True [✓ N/A] False

offset: True [✓ N/A] False

▶ Aggregation ▶ Other Keywords

(a) Screenshot of the Query builder, showing a few of the properties (with "Solution Modifiers" unfolded)


```
SELECT * FROM (
--
SELECT "id","origin","queryStr","select","parseError","hypertreeWidth"
FROM "Queries"
WHERE
  "select" = true AND
  "parseError" = false AND
  "hypertreeWidth" = 2
LIMIT 10 OFFSET 0
--
) AS q LEFT JOIN LATERAL (
  SELECT count(id) as d_count FROM "Duplicates" AS d
  WHERE q.id = "copyOfId"
) AS d ON true;
```

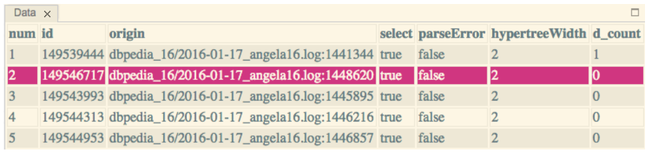
(b) Automatically generated SQL query (this one fetches the SPARQL select-queries with hypertreewidth two)

Figure 2: Query Builder and SQL text editor

- its results are shown in the query result table display (shown in Figure 3(a));
- the first result is displayed in the SPARQL text display (shown in Figure 3(b)) and
- the first result is visualized in the query visualizer (shown in Figure 3(c) and 3(d)).

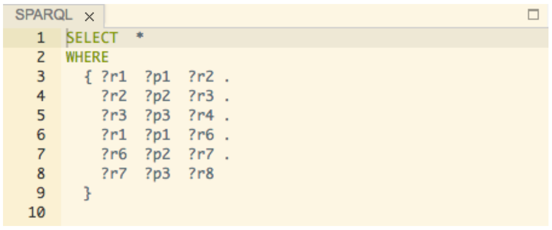
The entries in the result table display are clickable, so the user can immediately select a query she is interested in (e.g., the second entry in Figure 3(a)). When clicking a query, it is shown in the SPARQL editor and visualized (e.g., Figure 3(b)–3(d)) show the highlighted query from Figure 3(a) and two different visualizations). Furthermore, the query visualiser has links for going to the previous and next query. Additional properties of the currently displayed query can be shown in a "Details" panel (for which we did not provide a screenshot).

Figure 3(b) shows a query from the DBPedia 2016 (Jan. 17th) log file, corresponding to one of the results of the SQL query in Figure 2(b) on our database, fetching the queries of hypertreewidth two. The visualizer automatically renders a visualization of the query and can be configured to render the graph- and hypergraph structure. For the graph structure, we only consider the subject-object parts of SPARQL triples (and ignore the "predicate" part). Although the graph structure of a query is usually rather simple, it is often not sufficient to convey the full complexity of the query. For instance, the graph in Figure 3(c) ignores the variables ?p1, ?p2, and ?p3 from Figure 3(b).

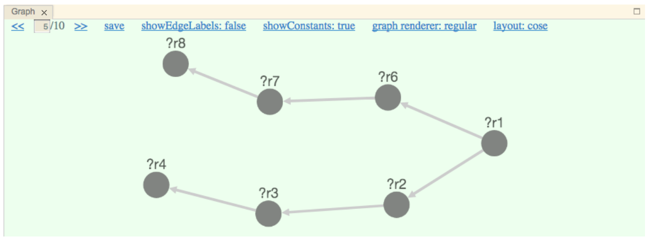


num	id	origin	select	parseError	hypertreeWidth	d_count
1	149539444	dbpedia_16/2016-01-17_angela16.log:1441344	true	false	2	1
2	149546717	dbpedia_16/2016-01-17_angela16.log:1448620	true	false	2	0
3	149543993	dbpedia_16/2016-01-17_angela16.log:1445895	true	false	2	0
4	149544313	dbpedia_16/2016-01-17_angela16.log:1446216	true	false	2	0
5	149544953	dbpedia_16/2016-01-17_angela16.log:1446857	true	false	2	0

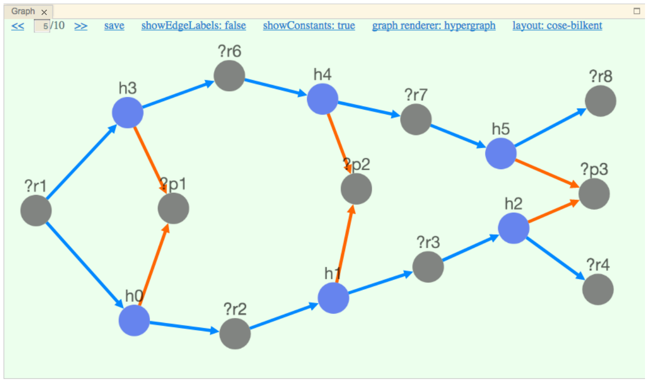
(a) Query result table (second entry is highlighted).


```
SELECT  *
WHERE
  { ?r1  ?p1  ?r2 .
    ?r2  ?p2  ?r3 .
    ?r3  ?p3  ?r4 .
    ?r1  ?p1  ?r6 .
    ?r6  ?p2  ?r7 .
    ?r7  ?p3  ?r8
  }
```

(b) One of the queries in our log (in the SPARQL editor).


<< 5/10 >> save showEdgeLabels: false showConstants: true graph renderer: regular layout: cose

(c) Visualization of the graph of Figure 3(b)'s query.

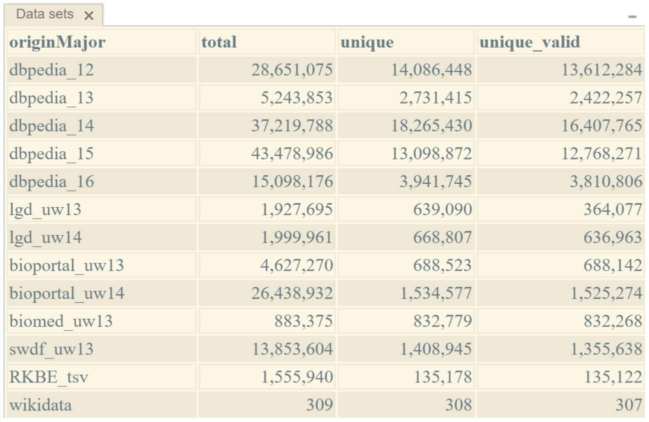

<< 5/10 >> save showEdgeLabels: false showConstants: true graph renderer: hypergraph layout: cose-bilkent

(d) Visualization of the hypergraph of Figure 3(b)'s query.

Figure 3: Partial screenshots of our query viewer and visualizer for an example query.

The hypergraph structure captures this complexity more accurately. Since we only had graph render libraries at our disposal, we display a hyperedge $\{s, p, o\}$ coming from SPARQL triple (s, p, o) as a new node h (representing the identity of the hyperedge) which we connect to nodes s, p, and o. We use the edges $s \longrightarrow h$ and $h \longrightarrow o$ (in blue) and $h \longrightarrow p$ (in orange). Figure 3(d) has such a visualization for the SPARQL query in Figure 3(b).

The visualizer currently uses several graph layout algorithms (cose, cose-bilkent, concentric, breadthfirst, grid, and circle) and can readily switch between them. This gives users a quick idea of the query's structure.



originMajor	total	unique	unique_valid
dbpedia_12	28,651,075	14,086,448	13,612,284
dbpedia_13	5,243,853	2,731,415	2,422,257
dbpedia_14	37,219,788	18,265,430	16,407,765
dbpedia_15	43,478,986	13,098,872	12,768,271
dbpedia_16	15,098,176	3,941,745	3,810,806
lgd_uw13	1,927,695	639,090	364,077
lgd_uw14	1,999,961	668,807	636,963
bioportal_uw13	4,627,270	688,523	688,142
bioportal_uw14	26,438,932	1,534,577	1,525,274
biomed_uw13	883,375	832,779	832,268
swdf_uw13	13,853,604	1,408,945	1,355,638
RKBE_tsv	1,555,940	135,178	135,122
wikidata	309	308	307

Figure 4: Data sets panel from the GUI, showing the currently loaded queries.

Finally, we provide a datasets panel (shown in Figure 4), which shows general statistics of the data that is currently in the database, ordered by origin. The panel contains four columns: name of the dataset (“originMajor”), total number of queries (“total”), number of unique queries (“unique”), and finally the number of unique queries that can be parsed (“unique_valid”). Duplicates are tested globally, so we may classify a query from dbpedia_15 as a duplicate if it already occured in dbpedia_12.

Of course, the GUI is such that it can show (and resize) all these panels at the same time, in order to give the user a complete overview. It also allows to hide the panels the user is currently not interested in. The panels can be used as stacked tabs or as split views with advanced layout capabilities found in IDEs.

3 DEMO OVERVIEW

We believe that many visitors of the demo will be interested to see how queries from actual SPARQL log files (DBPedia, BioPortal, LGD, OpenBioMed, Semantic Web Dog Food, British Museum) look like, since many of these log files are not publically available. Since DARQL is very flexible, and immediately shows visualization such as in Figures 3(c) and 3(d), we can do this interactively with visitors of the demo. Nevertheless, we will also prepare the following specific scenarios to stimulate such interaction.

Search for Complex Queries. Besides searching 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 SPARQL log files coming from disparate SPARQL endpoints, DARQL lets the users access the lineage of the queries under inspection and have a perception of what logs contain queries with certain complex characteristics.

Shape-Driven Exploration. Here we start by selecting a specific shape (e.g., “star”) and show visitors on the visualizer how star-shaped queries in the log files actually look like. This gives an impression on the size, complexity, branching, and diameter of such queries. The tool supports many different shapes to start from, among which star, tree, chain, forest, flower, and cycle. We can also start from a given hypertreewidth. (Queries with hypertreewidth three are already quite complex and rare in the query logs.)

In fact we already used the front-end extensively for our study of shapes in [1]: we gradually implemented more and more shape tests and then visually inspected queries that were not classified by any known shapes. So, this part of the tool has already been heavily used behind-the-scenes in our own research.

Getting Statistics. We will demonstrate how statistics such as “How many of the SELECT queries are conjunctive queries?” or “How many of the construct-queries do a non-trivial insertion?” can be computed. For every query that can be constructed with the query builder, we can toggle if the tool should count the number of the results, or if it should produce a sample of the answers. Therefore, such statistics can be easily computed.

Most Popular Queries. Statistics on the logs can be easily computed and lead to identify for instance the most (or the least) occurring queries in the entire corpus or in a single log file or data source. We can thus access the most (or less) popular queries in the logs with a breakdown view on each individual log file, on each individual data source (e.g., Wikidata, DBPedia, BioPortal etc.) or on the entire corpus.

Expert Search in SQL. The predefined user interface only generated SQL queries to the database that test conjunctions of conditions. Using the direct SQL interface, we show that if the user wants, also more advanced conditions can be queried. For example, one can search the union of all queries that have a minimum size and those that have cycles.

Furthermore, it is possible to explore queries in the context of time. Although only some logs have timestamps, most log files have names that indicate a date. Figure 3(a) shows five queries from January 17th, 2016, for example. Using the SQL editor, queries coming from a specific date (or month, or year) can be found as well. With this date, we could perform complex interesting queries with a temporal aspect. For instance, we could inspect how many queries were submitted on the same day, or try to find days or time spans that have the most or least queries. Or, we could calculate the time span (first and last occurrence) for duplicates of a query. A user can design her own very complex log searches by formulating them as queries in SQL.

To conclude, we want to give visitors of the demo the opportunity to browse in our repository consisting of 180 million SPARQL queries, gathered from 2007 to 2017 and ranging over a wide range of sources. We also want to advertise DARQL itself as a useful tool for analysing SPARQL logs. We believe that the tool is indeed useful for analyzing large logs. The demo will show that most queries run very fast, i.e., faster than a second.

REFERENCES

[1] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An Analytical Study of Large SPARQL Query Logs. *PVLDB* 11, 2 (2017), 149–161.
[2] Mark Kaminski and Egor V. Kostylev. 2016. Beyond Well-designed SPARQL. In *International Conference on Database Theory (ICDT).* 5:1–5:18.
[3] Francois Picalausa and Stijn Vansummeren. 2011. What Are Real SPARQL Queries Like? In *SWIM.* ACM, New York, NY, USA, Article 7, 6 pages.
[4] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. LSQ: The Linked SPARQL Queries Dataset. In *International Semantic Web Conference (ISWC).* 261–269.