An Analytical Study of Large SPARQL Query Logs

Angela Bonifati, Wim Martens, Thomas Timm

Noname manuscript No.
(will be inserted by the editor)

An Analytical Study of Large SPARQL Query Logs

Angela Bonifati · Wim Martens · Thomas Timm

the date of receipt and acceptance should be inserted later

Abstract With the adoption of RDF as the data model for Linked Data and the Semantic Web, query specification from end-users has become more and more common in SPARQL endpoints. In this paper, we conduct an in-depth analytical study of the queries formulated by end-users and harvested from large and up-to-date structured query logs from a wide variety of RDF data sources. As opposed to previous studies, ours is the first assessment on a voluminous query corpus, spanning over several years and covering many representative SPARQL endpoints. Apart from the syntactical structure of the queries, that exhibits already interesting results on this generalized corpus, we drill deeper in the structural characteristics related to the graph and hypergraph representation of queries.

We outline the most common shapes of queries when visually displayed as undirected graphs, characterize their tree width, length of their cycles, maximal degree of nodes, and more. For queries that cannot be adequately represented as graphs, we investigate their hypergraphs and hypertree width. Moreover, we analyze the evolution of queries over time, by introducing the novel concept of a streak, i.e., a sequence of queries that appear as subsequent modifications of a seed query. Our study offers several fresh insights on the already rich query features of real SPARQL queries formulated by real users, and brings us to draw a number of conclusions and pinpoint future directions for SPARQL query evaluation, query optimization, tuning, and benchmarking.

A. Bonifati
Lyon 1 University
Lyon, France

W. Martens
University of Bayreuth
Bayreuth, Germany

T. Timm
University of Bayreuth
Bayreuth, Germany

1 Introduction

As more and more data is exposed in RDF format, we are witnessing a compelling need from end-users to formulate more or less sophisticated queries on top of this data. SPARQL endpoints are increasingly used to harvest query results from available RDF data repositories. But how do these end-user queries look like? As opposed to RDF data, which can be easily obtained under the form of dumps (DBpedia and Wikidata dumps [45,46, 51]), query logs are often inaccessible, yet hidden treasures to understand the actual usage of these data. In this paper, we investigate a large corpus of query logs from different SPARQL endpoints, which spans over several years (2009–2017). In comparison to previous studies on real SPARQL queries [3,21,36,41,42], which typically[1] investigated query logs of a single source, we consider a multi-source query corpus that is two orders of magnitude larger. Furthermore, our analysis goes significantly deeper. In particular, we are the first to do a large-scale analysis on the topology of queries, which has seen significant theoretical interest in the last decades (e.g., [14,18,20]) and is now being used for state-of-the-art structural decomposition methods

[1] The exception is [21], where logs from the Linked SPARQL Queries (LSQ) dataset were studied, combining data from four sources (from 2010 and 2014) that we also consider.

for query optimization [1,2,26]. As a consequence, ours is the first analytical study on real (and most recent) SPARQL queries from a variety of domains reflecting the recent advances in theoretical and system-oriented studies of query evaluation.

Our paper makes the following contributions. Apart from classical measures of syntactic properties of the investigated queries, such as their keywords, their number of triples, and operator distributions, which we apply to our new corpus, we also mine the usage of projection in queries and subqueries in the various datasets. Projection indeed is the cause of increased complexity (from PTIME to NP-Complete) of the following central decision problem in query evaluation [13,8,30]: Given a conjunctive query Q, a database D, and a candidate answer a, is a an answer of Q on D?

We then proceed by considering queries under their graph and hypergraph structures. Such structural aspects of queries have been investigated in the database theory community for over two decades [18] since they can indicate when queries can be evaluated efficiently. Recently, several studies on new join algorithms leverage the hypergraph structure of queries in the contexts of relational and RDF query processing [1,26]. Theoretical research in this area traditionally focused on *conjunctive queries (CQs)*. For CQs, we know that tree-likeness of their structure leads to polynomial-time query evaluation [18]. For larger classes of queries, the topology of the graph of a query is much less informative. For instance, if we additionally allow SPARQL's Opt operator, evaluation can be NP-complete even if the structure is a tree [8]. For this reason, we focus our structural study on CQ-like queries.[2] We develop a shape classifier for such queries and identify their most occurring shapes. Interestingly enough, these queries have quite regular shapes. The overwhelming majority of the queries is acyclic (i.e., tree- or forest-shaped). We discovered that the cyclic queries mostly consist of a central node with simple, small attachments (which we call *flower*). In terms of tree- and hypertreewidth, we discovered that the cyclic queries have width two, up to a few exceptions with width three.

At this point we should make a note about interpretation of our results. Even though almost all CQ-like queries have (hyper-)treewidth one, we do not want to claim that queries of larger treewidth are not important in practice. The overwhelming majority of the queries we see in the logs are very small and simple, which we believe may be typical for SPARQL endpoint logs. For instance, the majority of the queries in our logs only use one triple. More precisely, this holds for over 52% of the valid queries and for over 58% of the unique valid queries. One of our data sets, `Wikidata17` is not a SPARQL endpoint log and we see throughout the paper that it has completely different characteristics.

In order to gauge the performances of cyclic and acyclic queries from a practical viewpoint, we have run a comparative analysis of chain and cycle queries synthetically generated with an available graph and query workload generator [4]. This experiment showed different behaviors of SPARQL query engines, such as Blazegraph and PostgreSQL with query workloads of CQs of increasing sizes (intended as number of conjuncts). It also lets us grasp a tangible difference between chain and cycle queries in either query engine, this difference being more pronounced for PostgreSQL. We may interpret this result as a lack of maturity of practical query engines for cyclic queries, thus motivating the need of specific query optimization techniques for such queries as in [1,26].

Finally, we deal with the problem of identifying sequences of similar queries in the query logs. These queries are then classified as gradual modifications of a seed query, possibly by the same user. We measure the length of such streaks in three log files from DBpedia. We conclude our study with insights on the impact of our analytical study of large SPARQL query logs on query evaluation, query optimization, tuning, and benchmarking.

This paper extends its conference version [11] as follows:

(1) We augment our corpus with 169M queries from the `DBpedia17` dataset, which was not considered before and let us almost double the size of our total valid queries.
(2) We perform all our analyses twice: once on the set of all *valid* and once on the set of all *unique valid* queries. The conference version only considered the unique valid queries. We note that the *valid* and *unique valid* logs give different insights about the data, which are complementary. The *valid* set gives an idea about the different types of queries in the logs and the *unique valid* set gives a better view on the queries and the workload that the SPARQL endpoint actually receives.[3]
(3) We extend our study to the Construct clause apart from Select and Ask queries considered in [11]. This means that the present study includes *all types of SPARQL queries with a well-defined semantics*. We also consider the Values keyword in the queries, because it is more frequent in our new corpus. The

[2] We consider extensions with Filter, Opt, and Values, but only in a way for which we know that tree-likeness of the query graph ensures the existence of efficient evaluation algorithms.

[3] For instance, as can be seen immediately in Figure 1, the DBpedia endpoint receives many more large queries than the *unique valid* logs lead us to suspect.

addition of Values leads to additional insights, such as a significant increase of cyclic queries in Table 7.

(4) On top of investigating well-designedness of queries (introduced by Perez et al. [40]), we also investigate *weak* well-designedness, a notion introduced by Kaminski and Kostylev [27], which is important because it also identifies a fragment of queries using And, Opt, and Filter that can be evaluated more efficiently than in the general case.

(5) We perform our shape analysis once for the graphs of queries with constants and once for the graphs without constants (i.e., only the variables). We believe that the shapes of queries with constants can be interesting for practitioners working on query evaluation and optimization. The shapes of queries without constants are usually considered in theoretical research on query evaluation, i.e., the treewidth and hypertreewidth of queries is usually only considered for the graph of the queries containing only the variables.

(6) We add more tests to the shape analysis, which give researchers a much more precise idea of the shape of queries. For instance, we investigate specific measures on the characteristics of the most common shapes, such as the longest path, the size of the maximal degree vertex, the number of high degree vertices and for cyclic queries the cycle lengths.

(7) We extend the hypergraph analysis with an analysis of *free-connex acyclicity*. This measure is very important in theory and practice, since it characterizes the conjunctive queries for which efficient algorithms exist for enumerating their output [6,24] (under standard complexity theory assumptions).

(8) We analyse the number of *tree pattern queries* in the query logs. Tree pattern queries or twig queries were heavily researched in the context of XML query languages and, due to their modal nature, can also be used for querying graphs [15,31]. We discover that they are quite prominent in the logs.

(9) Due to the additional queries, we obtain 404,721 property paths from unique queries (compared to 247,404 in [11]). Still, we manage to completely classify all these property paths in 35 types of expressions. (We only needed 21 types of expressions in [11].) Since property paths are a challenging issue in SPARQL queries and graph database queries in general [10], we believe this data to be very useful for developers of graph database engines.

We conclude the paper with observations and insights about further analyses on query logs.

Related Work. Whereas several previous studies have focused on the analysis of real SPARQL queries, they have mainly investigated statistical features of the queries, such as occurrences of triple patterns, types of queries, or query fragments [3,21,36,42]. The only early study that investigated the relationship between structural features of practical queries and query evaluation complexity has been presented in [41]. However, they focus on a limited corpus (3M queries from DBpedia 2010) and in that sense their findings cannot be generalized. Our work moves onward by precisely characterizing the occurrences of conjunctive and non-conjunctive patterns under the latest complexity results, by performing an accurate shape analysis of the queries under their (hyper)graph representation and introducing the evolution of queries over time. USEWOD and DBpedia datasets have also been considered in [3]. It takes into account the log files from DBpedia and SWDF reaching a total size of 3M. The work mainly investigates the number of triples and joins in the queries. Based on the observation of [39] that SPARQL graph patterns are typically chain- or star-shaped, they also look at their occurrences. They found very scarce chains and high coverage of almost-star-shaped graph patterns, but they do not characterize the latter. To the best of our knowledge, we are the first to carry out a comprehensive shape analysis on such a large and diverse corpus of SPARQL queries.

A query analysis and clustering of DBpedia SPARQL queries has been performed in [37] in order to build a set of prototypical benchmarking queries. Query logs have been inspected in a user study in [23] to understand whether facts that are queried together provide intra-fact relatedness in the Linked Open Data graph. The objectives of both papers are different from the one pursued in our work.

Large collections of Wikidata queries have been analyzed recently in [32,9], which focused on basic characteristics of queries related to their usage in the Wikidata query service and spanning from SPARQL feature prevalence and correlation to annotations and language distributions. They also do a classification of the queries in their corpus into robotic and organic, which would not be possible in our case since our logs lacks the information about browser- and machine-generated traffic. However, our analysis significantly differs from theirs since they do not study in-depth characteristics of the queries reflecting complexity classes, involving query shapes and property paths, along with the evolution of streaks, as we do in this paper.

2 Data Sets

Our data set has a total of 350,089,005 queries, which were obtained as follows. We obtained the 2013–2016

Table 1 Sizes of query logs in our corpus.

Source	*Total #Q*	*Valid #Q*	*Unique #Q*
DBpedia9-12	28,651,075	27,622,233	13,437,966
DBpedia13	5,243,853	4,819,837	2,628,000
DBpedia14	37,219,788	33,996,486	17,217,416
DBpedia15	43,478,986	42,709,781	13,253,798
DBpedia16	15,098,176	14,687,870	4,369,755
DBpedia17	169,110,041	164,297,723	34,440,636
LGD13	1,927,695	1,531,164	357,843
LGD14	1,999,961	1,951,973	628,640
BioP13	4,627,270	4,624,449	687,773
BioP14	26,438,932	26,404,716	2,191,151
BioMed13	883,375	882,847	27,030
SWDF13	13,853,604	13,670,550	1,229,759
BritM14	1,555,940	1,545,643	135,112
Wikidata17	309	308	308
Total	350,089,005	338,745,580	90,605,187

USEWOD query logs, some additional DBpedia query logs for 2013, 2014, 2015, 2016, and 2017 directly from Openlink[4], the 2014 British Museum query logs from LSQ[5], and we crawled the user-submitted example queries from Wikidata[6] in February 2017. These log files are associated with 7 different data sources from various domains: DBpedia, Semantic Web Dog Food (SWDF), LinkedGeoData (LGD), BioPortal (BioP), OpenBioMed (BioMed), British Museum (BritM), and Wikidata.

Table 1 gives an overview of the analyzed query logs, along with their main characteristics. Since we obtained logs for DBpedia from different sources, we proceeded as follows. `DBpedia9-12` contains the DBpedia logs from USEWOD'13, which are query logs from 2009–2012. All other `DBpedia'X` sets contain the query logs from the year `'X`, be it from USEWOD or from Openlink.[7]

Compared to the conference version of this article [11], we have obtained 169,110,041 new queries from Openlink, which is reflected in the `DBpedia17` dataset. Some of the other data sets are slightly larger than in the conference version, due to an issue with the parser, which we fixed. In some cases, the parser would have an internal error and the query would not even show up in our total count.

We prepared the logs for analysis as follows. We first cleaned the logs, since some contained entries that were not queries (e.g., http requests). In the following we only report on the actual SPARQL queries in the logs. For each of the logs, the table summarizes the total number of queries (*Total*) and the number of queries that we could parse using Apache Jena 3.7.0 (*Valid*). From the latter set, we removed duplicate queries after whitespace normalization, resulting in the unique queries that we could parse (*Unique*). In the remainder of the paper, we present results on both *Valid* and *Unique* data sets. In [11] we reported the results for the *Unique* corpus only. Adding the *Valid* data set is important for improving our understanding of the query logs though: while the *Unique* data set gives us an idea of the different types of queries that appear in the logs, the *Valid* data set gives a better idea of the queries and the workload that the SPARQL endpoints actually receive. In summary, our corpus of query logs contains the latest blend of USEWOD and Openlink DBPedia query logs (the latter providing 51M more queries in the period 2013–2016 than the USEWOD corpus, and 169M more for 2017), plus BritM and Wikidata queries. We are not aware of other existing studies on such a large and up-to-date corpus. Finally, although the online Wikidata example queries (Feb 13th, 2017) are a manually curated set, there was one query that we could not parse.[8]

Throughout the article, we will use the following notation to discuss results on the *Valid* and *Unique* data sets. Whenever we report a number or a percentage in the format X (Y), the number X refers to the *Valid* and the number Y to the *Unique* set of queries. This notation allows the reader to stay informed about the queries that the endpoint actually receives (*Valid*) and about those without duplicates in this set (*Unique*).

The query logs we received are anonymized in the sense that they do not contain IP addresses, precise time stamps, or user agents. Time stamps are typically either completely absent, or rounded to an hour. (In some of the logs, all time stamps are set to 3:00.) This means, in particular, that these logs do not allow a classification into *robotic* and *organic* queries, as was done by Bielefeldt et al. [9] and Malyshev et al. [32].

In the total data set, 16,639,701 (2,978,945) queries, or 4.91% (3.29%) of the logs do not have a body. All these queries are Describe queries and almost exclusively occur in `DBpedia14`–`DBpedia17`. To be more precise, 99.47% (97.22%) of the Describe queries do not

[4] `http://www.openlinksw.com`

[5] `http://aksw.github.io/LSQ/`

[6] `https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples`

[7] We discovered that we received three log files from USEWOD as well as from Openlink, in the sense that only the hash values used for anonymisation were different. These duplicate log files were deleted prior to all analysis and are not taken into account in Table 1.

[8] The query was called "Public Art in Paris" and was malformed (closing braces were missing and it had a bad aggregate). It was still malformed on June 29th, 2017.

have a body. We therefore conduct some of our analyses only on Select, Ask, and Construct queries.

3 Preliminaries

We recall some basic definitions on RDF and SPARQL [40,41]. We closely follow the exposition of [41].

RDF. RDF data consists of a set of triples $\langle s, p, o\rangle$ where we refer to s as *subject*, p as *predicate*, and o as *object*. According to the specification, s, p, and o can come from pairwise disjoint sets $\mathcal{I}$ (*IRIs*), $\mathcal{B}$ (*blank nodes*), and $\mathcal{L}$ (*literals*) as follows: $s \in \mathcal{I} \cup \mathcal{B}$, $p \in \mathcal{I}$, and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. For this paper, the precise definition of IRIs, blank nodes, and literals is not important. The most important thing to know is that we treat blank nodes similar to *variables*, which we discuss later.

SPARQL. For our purposes, a *SPARQL query* Q can be seen as a tuple of the form

$$(\textit{query-type}, \textit{pattern } P, \textit{solution-modifier}).$$

We now explain how such queries work conceptually. The central component is the *Pattern* P, which contains patterns that are matched onto the RDF data. The result of this part of the query is a multiset of mappings that match the pattern to the data.

The *solution-modifier* allows aggregation, grouping, sorting, duplicate removal, and returning only a specific window (e.g., the first ten) of the multiset of mappings returned by the pattern. The result is a list L of mappings.

The *query-type* determines the output of the query. It is one of four types: Select, Ask, Construct, and Describe. Select-queries return projections of mappings from L. Ask-queries return a Boolean and answer true iff the pattern P could be matched. Construct queries construct a new set of RDF triples based on the mappings in L. Finally, Describe queries return a set of RDF triples that describes the IRIs and the blank nodes in L. The exact output of Describe queries is implementation-dependent. Such queries are meant to help users explore the data. Compared to [41], we allow more solution modifiers and more complex patterns, as explained next.

Patterns. Let $\mathcal{V} = \{?x, ?y, ?z, ?x_1, \ldots\}$ be an infinite set of variables, disjoint from $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$. As in SPARQL, we always prefix variables by a question mark. A *triple pattern* is an element of $(\mathcal{I} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$. A *property path* is a regular expression over the alphabet $\mathcal{I}$. A *property path pattern* is an element of $(\mathcal{I} \cup \mathcal{B} \cup \mathcal{V}) \times pp \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$, where pp is a property path. A *SPARQL pattern* is an expression generated from the following grammar:

$$\begin{aligned} P ::= \quad & t \mid pp \mid Q \mid P_1 \text{ And } P_2 \mid P \text{ Filter } R \\ & \mid P_1 \text{ Union } P_2 \mid P_1 \text{ Opt } P_2 \\ & \mid \text{Graph } iv\ P \mid \text{Values } tup\ T \end{aligned}$$

Here, t is a triple pattern, pp is a property path pattern, Q is again a SPARQL query, R is a so-called *SPARQL filter constraint*, and $iv \in \mathcal{I} \cup \mathcal{V}$. We note that property paths (pp) and subqueries (Q) in the above grammar are new features since SPARQL 1.1. SPARQL filter constraints R are built-in conditions which can have unary predicates, (in)equalities between variables, and Boolean combinations thereof. The keyword Values binds a tuple *tup* to values in a given table T. We refer to the SPARQL 1.1 recommendation [22] and the literature [40] for the precise syntax of filter constraints and the semantics of SPARQL queries. We write vars(P) to denote the set of variables occurring in P.

We illustrate by example how our definition corresponds to real SPARQL queries. The following query comes from WikiData ("Locations of archaeological sites", from [45]).

```
SELECT ?label ?coord ?subj
WHERE
{?subj wdt:P31/wdt:P279* wd:Q839954 .
 ?subj wdt:P625 ?coord .
 ?subj rdfs:label ?label filter(lang(?label)="en")}
```

The query uses the property path `wdt:P31/wdt:P279*`, literal `wd:Q839954`, and triple pattern `?subj wdt:P625 ?coord`. It also uses a filter constraint. In SPARQL, the And operator is denoted by a dot (and is sometimes implicit in alternative, even more succinct syntax). The Select query will return all bindings of `?label`, `?coord`, and `?subj` for which the body can be satisfied. If we would turn it into an Ask query, i.e., replace the entire with the keyword `ASK`, it would return true if and only if the Select query would return at least one output.

The following Construct query from WikiData [45] constructs a new RDF graph related to "asthma" (literal `wd:Q35869`), by recording the respective qualifiers and their provenance information if available as Opt edges.

```
CONSTRUCT {
  wd:Q35869 ?p ?o . ?o ?qualifier ?f .
  ?o prov:wasDerivedFrom ?u . ?u ?a ?b .}
WHERE {
  wd:Q35869 ?p ?o . OPTIONAL {?o ?qualifier ?f .}
  OPTIONAL {?o prov:wasDerivedFrom ?u . ?u ?a ?b .}}
```

Finally, we define conjunctive queries, which are a central class of queries in database research and which we will build on in the remainder of the paper. In the context of SPARQL, we define them as follows.

Definition 1 A *conjunctive query (CQ)* is a SPARQL pattern that only uses the triple patterns and the operator And.

4 Shallow Analysis

In this section we investigate simple syntactical properties of queries.

4.1 Keywords

A basic usage analysis of SPARQL features was done by counting the keywords in queries. The results are in Table 2.[9]

The table contains four blocks: types of queries, solution modifiers, SPARQL algebra operators, and aggregation operators. In each of the blocks, we sorted the operators by their number of occurrences in the *Valid* data set.

The first block in Table 2 describes the type of queries. In total, 91.96% (88.22%) of the queries are Select queries, 4.94% (3.38%) Describe queries, 2.44% (6.56%) are Ask queries, and 0.67% (1.84%) Construct queries. There are, however, tremendous differences between the data sets. BioMed13 has less than 3.47% (12.83%) Select queries and almost 94% (85%) Describe queries, whereas LGD13 has 17% (28%) Select queries and almost 81% (71%) Construct queries.

Even within the same kind of data, we see significant differences. DBpedia16 has 85% (62%) Select queries (and 12.1% (34%) Describe queries), whereas DBpedia15 has 92% (81.5%) Select queries and 4% (11.5%) Ask queries. The other DBpedia data sets have over 87.5% Select queries. DBpedia17 has 91% (88%) Select queries, 2.1% (9.1%) Ask queries and 5.8% (1.4%) Describe queries.

The second block in Table 2 contains solution modifiers, ordered by their popularity.[10] Looking into the specific data sets, we see the following things stand out. Almost all 89% (97%) of BritM14 queries use Distinct. This is similar, but to a lesser extent in BioP13 (96% (82%)) and BioP14 (92%(68%)). In DBpedia we again see significant differences. From '12 to '17, we have 21% (18%), 7% (8%), 16% (11%), 20% (38%), 6% (8%) and 26% (52%) of queries with Distinct respectively.

Limit is used most widely in SWDF13 (48 (47%)), in LGD13 (59% (17%)) and LGD14 (54 (41%)). The most prevalent data sets for queries with Offset are LGD14 (30% (38%)), LGD13 (52%(13%)), and DBpedia13 (10% (12%)).

Order By is used by far the most in Wikidata (44%), which may be due to the case that Wikidata17 is not a query log, but a Wiki page that contains cherry-picked and user-submitted queries. These queries are intended to showcase system's behavior or highlight features of the Wikidata data set and should therefore produce a nice output. The other data sets are true query logs, which may therefore also contain the "development process" of queries: users start by asking a query and gradually refine it until they have the one they want. (We come back to this in Section 10).

The third block has keywords associated to SPARQL algebra operators that occur in the body. We see that Filter, And, Union, and Opt are quite common.[11] The next commonly used operator is Graph but, looking closer at our data, we see that 96% (78%) and 85% (40%) of the queries using Graph originate from BioP13 and BioP14. The use of Filter ranges from 63% (58%) for DBpedia13 to 0.7% (3%) or less for BioMed13 and BioP13, respectively.

The fourth block has aggregation operators. We were surprised that these operators are used so sparsely, even though aggregates are only supported since SPARQL 1.1 (March 2013) [22]. In all data sets, each of these operators was used in 3% or less of the Unique queries, except for LGD14 (31% with Count), DBpedia17 (11% with Group By) and Wikidata17 (30% with Group By). We see a higher relative use of aggregation operators in Wikidata17 than in the other sets, which we again believe is due to the fact that the Wikidata17 set is not a query log.

Overall, when we compare the *Unique* and *Valid* logs, it is striking that the relative occurrences of the four main SPARQL algebra operators Filter, And, Union, and Opt all decrease when eliminating duplicate queries.

4.2 Number of Triples in Queries

In order to measure the size of the queries belonging to the datasets under study, we have counted the total number of triples of the kind $\langle s, p, o \rangle$ contained in Select, Ask and Construct queries. In this experiment, we merely counted the number of triples contained in each query without further investigating the possible relationships among them (such as join conditions, unions etc.), which are studied in the remainder of the paper.

[9] We also investigated the occurrence of other operators (Service, Bind, Assign, Data, Dataset, Sample, Group Concat), each of which appeared in less than 1% of the queries. We omit them from the table for succinctness.

[10] The remaining solution modifier, Reduced, was only found in 6,126 (1,149) queries.

[11] Conjunctions in SPARQL are actually denoted by "." or ";" for brevity, but we group them under "And" in this paper for readability.

Table 2 Keyword count in queries

Element	*Absolute V*	*Relative V*	*Absolute U*	*Relative U*
Select	311,496,923	91.96%	79,929,422	88.22%
Describe	16,727,191	4.94%	3,061,636	3.38%
Ask	8,265,673	2.44%	5,943,216	6.56%
Construct	2,255,793	0.67%	1,670,913	1.84%
Distinct	96,055,447	28.36%	29,973,911	33.08%
Limit	46,442,970	13.71%	17,043,706	18.81%
Offset	8,651,005	2.55%	4,112,839	4.54%
Order By	3,481,015	1.03%	1,609,921	1.78%
Filter	148,681,968	43.89%	34,609,372	38.20%
And	129,524,653	38.24%	26,737,378	29.51%
Opt	107,447,875	31.72%	13,119,429	14.48%
Union	85,024,759	25.10%	15,761,764	17.40%
Graph	27,556,055	8.13%	1,523,675	1.68%
Values	7,595,583	2.24%	5,086,033	5.61%
Not Exists	2,527,452	0.75%	1,096,099	1.21%
Minus	2,199,152	0.65%	1,664,359	1.84%
Exists	13,965	0.00%	7,832	0.01%
Group By	9,100,381	2.69%	3,887,216	4.29%
Count	924,474	0.27%	653,756	0.72%
Having	197,463	0.06%	40,401	0.04%
Avg	7,714	0.00%	731	0.00%
Min	7,040	0.00%	3,749	0.00%
Max	6,504	0.00%	3,796	0.00%
Sum	2,768	0.00%	785	0.00%

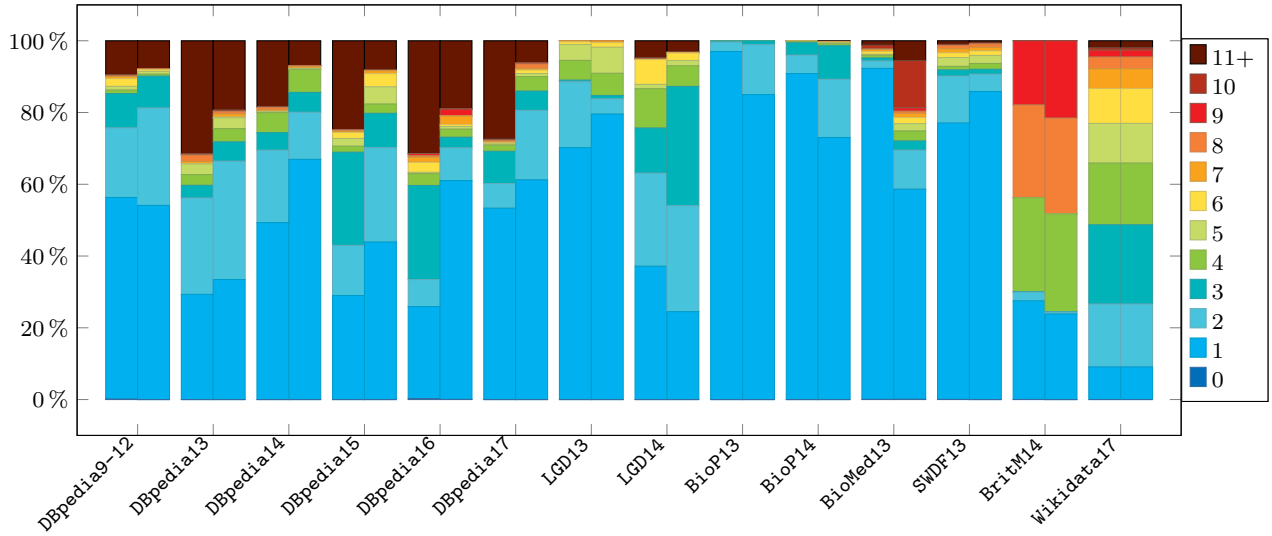


Fig. 1 Percentages of queries exhibiting different number of triples (in colors) for each dataset for Valid (left hand side of each bar) and Unique queries (right hand side of each bar).

We focus on Select, Ask and Construct queries as opposed to [11], which analysed Select and Ask on their corpus. We discard the Describe statements, which have an implementation-dependent semantics. [12]

The plot in Figure 1 illustrates how queries containing 0 to 11+ triples are distributed over the Select, Ask and Construct queries in each of the data sets. A first observation that we can draw from Figure 1 is that for the majority of the datasets, the queries with a low number of triples (from 0 to 2) have a noticeable share within the total amount of queries per dataset. Whereas these queries are almost the only queries present in the `BioP13` and `BioP14` datasets, they have the least concentration in `BritM14` and `Wikidata17`. The latter datasets have in fact unique characteristics, `BritM14` being a collection of queries with fixed templates and

[12] For instance, 95% (97%) of the Describe statements in our corpus do not have a body and therefore no triples.

Wikidata17 being the most diverse dataset of all, gathering queries of rather disparate nature that are representatives of classes of real queries issued on Wikidata. Finally, DBpedia9-12 until DBpedia17, along with LGD14 and BioMed13 are the datasets exhibiting the most complex queries with extremely high numbers of triples exceeding 10.

We should note that BioMed13 has almost 94% (87%) Describe queries. The numbers reported here only depict the remaining 6% (13%).

Overall, we observe that 63.62% (58.40%) of the Select, Ask and Construct queries in our corpus use at most one triple, 77.89% (90.16%) uses at most six triples, and 99.44% (98.35%) at most twelve triples. The largest queries we found came from DBpedia15 (209 and 211 triples) and BioMed13 (221 and 229 triples). In the new query logs of DBpedia17, the largest queries contain 207 and 209 triples.

If we compare the *Unique* and *Valid* query logs overall, we see that the *Valid* logs usually have more large queries than the *Unique* logs (sometimes quite significantly, e.g., in DBpedia17). This means that, in particular, the DBpedia SPARQL endpoint seems to receive significantly more large queries than what the results on *Unique* queries in [11] suggest, but also that there are many duplicates among these large queries.

4.3 Operator Distribution

In Table 2 we see that Filter, And, Union, Opt, and Graph are used fairly commonly in the bodies of Select-, Ask-, and Construct queries. We can notice that the numbers in Table 2 are generally compatible with those of our previous corpus in [11]. We can notice, however, a remarkable increase in the usage of Group By queries (from 0.3% to 4.29% in the new corpus).

We then investigated how these operators occur together. In particular, we investigated for which queries the body *only* uses constructs with these operators.[13]

The results are in Table 3, which has two kinds of rows. Each white row has, on its left, a set S of operators from $\mathcal{O} = \{\mathsf{Filter}, \mathsf{And}, \mathsf{Opt}, \mathsf{Graph}, \mathsf{Union}, \mathsf{Values}\}$ and, on its right, the amount of queries in our logs for which the body uses exactly the operators in S (and none from $\mathcal{O} \setminus S$). The value for *none* is the amount of queries that do not use any of the operators in $\mathcal{O}$ (including queries that do not have a body).

Conjunctive patterns with filters are considered to be an important fragment of SPARQL patterns, because they are believed to appear often in practice [39, 50]

Definition 2 A *conjunctive query with filters (CQ_F)* is a SPARQL pattern that only uses triple patterns and the operators And and Filter.

Our logs contain 50.51% (66.89%) $\mathrm{CQ_F}$ queries. Adding Opt to the $\mathrm{CQ_F}$ fragment would increase its relative size with 11.80% (7.20%) resulting in 62.31% (74.09%) our queries. (Similarly for Union, Graph and Values.) Table 3 classifies 95.07% (96.62%) of the Select, Ask and Construct queries in our corpus. The remaining queries either use other combinations from $\mathcal{O}$ 1.64% (2.79%) or use other features than those in $\mathcal{O}$ in their body 2.10% (3.61%) like Bind, Minus, subqueries, or property paths. A recurrent combination of features than those in $\mathcal{O}$ has been observed in the latest query logs (DBpedia17), in which Union and Values appear together in 1.30% (5.08%) of the queries, whereas they are mostly not existing in the other datasets.

When we compare the *Valid* with the *Unique* data sets, two changes stand out: Graph and the A,F,O,U fragment become much less common when duplicates are removed. For Graph, it seems that the BioPortal query logs are responsible, since these logs harbor almost all queries that use Graph. For the A,F,O,U fragment, we see that all DBpedia logs from 2013 on contain many duplicates of A,F,O,U queries. For instance, in the *Valid* DBpedia17 logs we have 25.87% A,F,O,U queries, but in the *Unique* DBpedia17 logs, this fragment only constitutes 6.06% of the queries.

4.4 Subqueries and Projection

Only 1309040 (575666) queries in our corpus use subqueries. The feature was most used in WikiData (9.74%), about an order of magnitude more than in any of the other data sets.

Projection plays a crucial role in the complexity of query evaluation. Many papers [8,30,27,40,41] define evaluation as the following question: *Given an RDF graph G, a SPARQL pattern P, and a mapping μ, is μ an answer to P when evaluated on G?* In other words, the question is to verify if a candidate answer μ is indeed an answer to the query. If P is a CQ, this problem is NP-complete if the queries use projection [13,8,30], but its complexity drops to PTIME if projection is ab-

[13] There is one exception: For Wikidata, we removed SERVICE subqueries before the analysis (which appears in approximately 200 of its queries and is used to change the language of the output).

Table 3 Sets of operators used in queries: And (A), Filter (F), Graph (G), Opt (O), Union (U), and Values (V)

Operator Set	*AbsoluteV*	*RelativeV*	*AbsoluteU*	*RelativeU*
none	107,285,016	33.32%	31,785,844	36.31%
A	15,106,778	4.69%	7,769,170	8.87%
F	30,679,572	9.53%	14,822,993	16.93%
A,F	9,583,490	2.98%	4,176,586	4.77%
CQF subtotal	162,654,856	50.51%	58,554,593	66.89%
O	2,921,810	0.91%	625,663	0.71%
A,O	3,436,987	1.07%	1,807,483	2.06%
F,O	7,115,439	2.21%	2,096,526	2.39%
A,F,O	24,512,799	7.61%	1,773,624	2.03%
CQF+O	+37,987,035	+11.80%	+6,303,296	+7.20%
U	8,533,645	2.65%	4,627,921	5.29%
A,U	1,627,742	0.51%	1,010,579	1.15%
F,U	627,559	0.19%	254,640	0.29%
A,F,U	1,824,697	0.57%	1,057,080	1.21%
CQF+U	+12,613,643	+3.92%	+6,950,220	+7.94%
V	151,078	0.05%	63,912	0.07%
A,V	207,180	0.06%	164,175	0.19%
F,V	2,497,572	0.78%	2,204,598	2.52%
A,F,V	142,211	0.04%	98,560	0.11%
CQF+V	+2,998,041	+0.93%	+2,531,245	+2.89%
G	26,288,960	8.16%	1,380,991	1.58%
A,G	391,433	0.12%	42,315	0.05%
F,G	876	0.00%	269	0.00%
A,F,G	34,418	0.01%	9,495	0.01%
CQF+G	+26,715,687	+8.30%	+1,433,070	+1.64%
A,F,O,U	67,026,601	20.81%	6,170,843	7.05%

sent [40,8,30].[14] Therefore, the use of projection has a huge influence of the complexity of query evaluation.

Surprisingly, we discovered that at least 9.1% (13.13%) of the queries use projection, which is significantly higher than what Picalausa and Vansummeren discovered in DBpedia logs from 2010 [41]. The 9.1% (13.13%) consists of 8.33% (11.88%) Select queries plus 0.76% (1.24%) Ask queries. Notice that the total number of Ask queries 2.44% (6.56%) is significantly higher, even though they just return a Boolean value and one would intuitively expect that almost all of them would use projection. The reason is that most Ask queries do not use variables: they ask if a concrete RDF triple is present in the data. Following the test for projection in Section 18.2.1 in the SPARQL recommendation [22], we classified these queries as not using projection.

Due to the use of the Bind operator or to the presence of subqueries, there was a number of queries (3.08% for Valid and 5.37% for Unique queries) where we could not determine if they use projection or not. Therefore the number of queries with projection lies between 9.1% and 12.18% for Valid queries (13.13% and 18.5% for Unique queries, respectively).

[14] This difference can be understood as follows: If the query tests the presence of a k-clique, then without projection we are given a k-tuple of nodes and need to verify if they form a k-clique. With projection, we need to solve the NP-complete k-clique problem.

5 Structural Analysis

SPARQL patterns of queries using only triple patterns and the operators And, Opt, and Filter (and, in particular, not using subqueries or property paths) received considerable attention in the literature (see, e.g., [40, 27,8,29,30]). We refer to such Select, Ask, or Filter patterns as *And/Opt/Filter patterns* or, for succinctness, *AOF patterns*. Our corpus has 200,641,891 (64,857,889) AOF patterns, which amounts to 62.31% (74.09%) of the Select, Ask, and Construct queries.

In Sections 6 and 7 we investigate the graph- and hypergraph structure of AOF patterns. The graph structure gives us a clear view on how such queries are structured and can tell us how complex such queries are to evaluate. For a significant portion of queries, however, the graph structure is not meaningful to capture their complexity (cf. Example 1) and we therefore need to turn to their hypergraph structure. Since the graph structure may be easier to understand and is often sufficient, we use the graph structure whenever we can.

We provide some background on the relationship between the (hyper)graph structure of queries and the complexity of their evaluation. Evaluation of CQs is NP-complete in general [13], but becomes PTIME if their *hypertree width* is bounded by a constant [20]. Here, the hypertree width measures how close the query is to a tree (the lower the width, the closer the query is to a tree). Several state-of-the-art join evaluation algorithms (e.g., [1,26]) effectively use the hypergraph structure of queries to improve their performance, even in the context of RDF processing [2]. We establish in Section 5.2 that there are significant performance differences in today's query engines, even when the hypertreewidth of queries just increases from one to two.

5.1 Graph and Hypergraph of a Query

We first make more precise what we mean by the graph and hypergraph of a query. An *(undirected) graph* G is a pair (V, E) where V is its (finite) set of nodes and E is its set of edges, where an edge e is a set of one or two nodes, i.e., $e \subseteq V$ and $|e| = 1$ or $|e| = 2$. A *hypergraph* $\mathcal{H}$ consists of a (finite) set of nodes $\mathcal{V}$ and a set of hyperedges $\mathcal{E} \subseteq 2^{\mathcal{V}}$, that is, a hyperedge is a set of nodes.

Most SPARQL patterns do not use variables as predicates, that is, they use triple patterns (s, p, o) where p is an IRI. We also allow $p \in$ vars if p is not used elsewhere in the query (in this case, p serves as a wildcard, possibly binding to a value that is returned to the output). We call such patterns *graph patterns*. Evaluation of graph patterns is tightly connected to finding embeddings of the graph representation of the query into the data.[15] We define the *triple graph* of graph pattern P to be the following graph: $E = \{\{x, y\}) \mid (x, \ell, y)$ is a triple pattern in P and $\ell \in \mathcal{I} \cup \mathcal{V}\}$ and $V = \{x \mid \{x, y\} \in E\}$.

Hypergraph representations can be considered for all AOF patterns. The *triple hypergraph of a SPARQL pattern* P is defined as $\mathcal{E} = \{X \mid$ there is a triple pattern t in P such that X is the set of blank nodes and variables appearing $t\}$ and $\mathcal{V} = \cup_{e \in E} e$.

For several types of queries, we will analyse the structure of their triple graph. However, the usage of some keywords of types of subqueries (notably, Filter and Values) can put additional constraints on the query that are not reflected in the triple (hyper)graph and we therefore need to augment it with additional (hyper)edges. We will call the resulting graphs the *canonical (hyper)graphs* of the queries. For CQs however, we define their canonical (hyper)graph to be equal to their triple (hyper)graph.

[15] In particular, it consists of finding embeddings of the directed and edge-labeled variant of the graph, but we omit the edge directions and -labels for simplicity. They do not influence the structure and cyclicity of graph patterns.

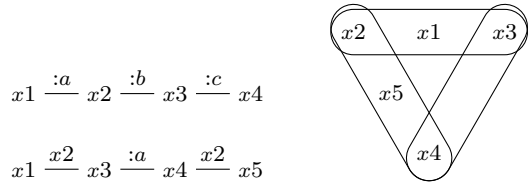


Fig. 2 Canonical graphs and hypergraph for queries in Example 1.

Example 1 Consider the following (synthetic) CQs:

```
ASK WHERE {?x1 :a ?x2 . ?x2 :b ?x3 . ?x3 :c ?x4}
ASK WHERE {?x1 ?x2 ?x3 . ?x3 :a ?x4 . ?x4 ?x2 ?x5}
```

Figure 2 (top left) depicts the canonical graph of the first query, which is a sequence of three edges. (We annotated the edges with their labels in the query to improve understanding.) The bottom left graph in Figure 2 shows why we do not consider canonical graphs for queries with variables on the predicate position in triples. The topological structure of this graph is a sequence of three edges, just as for the first query. This completely ignores the join condition on $?x2$. For this query, the canonical hypergraph in Figure 2 (right) correctly captures the cyclicity of the query.

5.2 Comparative Evaluation of Chain and Cycle Queries

We conducted a set of experiments aiming at comparing the execution times of conjunctive queries whose canonical graphs exhibit specific shapes. We have chosen chain and cycle queries in this empirical study. A *chain query (of length k)* is a CQ for which the canonical graph is isomorphic to the undirected graph with edges $\{x_0, x_1\}, \{x_1, x_2\}, \ldots, \{x_{k-1}, x_k\}$. (The first query in Example 1 is a chain query of length three.) A *cycle query (of length k)* is a CQ for which the canonical graph is isomorphic to $\{x_0, x_1\}, \ldots, \{x_{k-1}, x_0\}$. As an edge case, we also allow chains of length zero. Such chains consist either of a single node or no node at all. These shapes have been selected as representatives of the queries with hypertreewidth 1 and 2, respectively, and have also been used to compare the performances of join algorithms in other studies, e.g., [26].

In order to generate query workloads containing the aforementioned types of queries, we have used gMark [4], a publicly available[16] schema-driven generator for graph

[16] https://github.com/graphMark/gmark

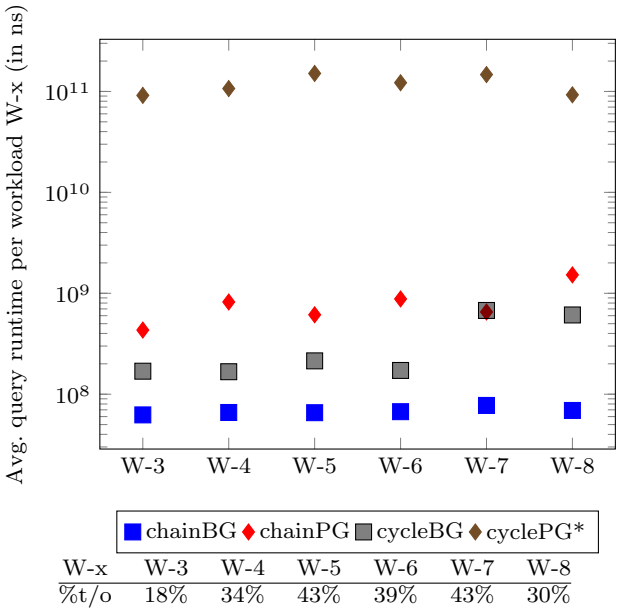
instances and graph queries. We tuned gMark to generate diverse query workloads, each containing 100 chain and cycle queries, respectively.[17] Each workload has been generated by using chains and cycles of different length varying from 3 to 8. In these experiments, we have considered and contrasted two opposite graph database systems, namely PostgreSQL [49], an open-source relational DBMS, and BlazeGraph [47], a high-performance SPARQL query engine powering the Wikimedia's official query service [51] and thus used for the official Wikidata SPARQL endpoint. We have run these experiments on 2-CPUs Intel Xeon E5-2630v2 2.6 GHz server[18] with 128GB RAM and running Ubuntu 16.04 LTS. We used PostgreSQL v.9.3 and Blazegraph v.2.1.4 for the experimental setup. We employed the Bib use case in the gMark configuration [4] for the schema of the generated graph (of size $100k$ nodes) and of the generated queries as well. We employed the query workloads in SQL and SPARQL as generated by gMark after elimination of empty unions (since gMark is geared towards generating UCRPQs) and of the keyword Distinct in the body of the queries. Since gMark allowed us to obtain mixed workloads of Select/Ask queries and we wanted to focus on one query type at a time, we manually replaced the Select clauses with compatible Ask clauses.

Figure 3 (top) depicts the average runtime (in ns, logscale) of our workloads of chain (cycle, resp.) queries with length from 3 to 8 on Blazegraph (BG) and PostgreSQL (PG). We can observe that the overall performance of BG is superior to that of PG. Indeed, in PG many cycles queries are timed out (after 300s per query) and we expect that the real overall performance of PG is even worse [19] than the results reported in Figure 3. Figure 3 (bottom) reports the reached timeouts for workloads of cycle queries of various sizes when executed in PG. It is worthwhile observing that for both systems the difference between average runtime of chain query workloads and cycle query workloads is non negligible, thus confirming that we cannot ignore the graph representation and the shape of queries. This experiment also motivated us to dig deeper in the shape analysis of our query logs, which we report in Section 6.

[17] We recall that gMark can generate queries of four shapes: chain, star, chain-star and cycle. We have thus cherry-picked chain queries as representatives of queries with hypertreewidth equal to 1.

[18] Every CPU has 6 physical cores and, with hyperthreading, 12 logical cores.

[19] in the case in which we let PG run beyond the time out and collect the new numbers.



W-x	W-3	W-4	W-5	W-6	W-7	W-8
%t/o	18%	34%	43%	39%	43%	30%

Fig. 3 Execution times (top) of diverse workload of chain/cycle queries (of length 3,4,5,6) on Blazegraph (BG) and Postgresql (PG). Number of timeouts per workload for CyclePG only (bottom). CyclePG times include t/o of 300s (per query).

5.3 Classes of Queries for (Hyper)graphs

We now discuss the classes of queries for which we will investigate their canonical graph- and hypergraph structures in Section 6. To the best of our knowledge, all the literature relating (hyper)graph structure of queries to efficient evaluation was done on AOF patterns. Here, we focus on fragments of AOF patterns, plus a mild extension, namely with additional Values-blocks. The simplest queries we consider are the CQs, which motivated the classical literature on query evaluation and hypertree structure [13,20]. We discovered that 61.00% (60.99%) of the AOF patterns are CQs.

Definition 3 A CQ is *suitable for graph analysis* if it is a graph pattern. For a CQ that is suitable for graph analysis, its *canonical graph* is defined as its triple graph. For every other CQ, its *canonical hypergraph* is defined as its triple hypergraph.

Next, we extend the above terminology for CQs with Filter, Opt, and Values. We only want to consider canonical (hyper)graphs for queries such that the relationship between efficient query evaluation and their (hyper)graph structure is still similar as for CQs. However, this requires some care, especially when considering Opt [8,40].

$\mathrm{CQ_F}$ patterns can be evaluated similarly to CQs, but we need discuss the fragment for which we will analyse the graph- structures. We say that a filter con-

straint R is *simple* if vars(R) contains at most two variables. (An almost identical class of queries was considered in [41].)

Definition 4 A CQ_F query is *suitable for graph analysis* if it is a graph pattern and all filter constraints are simple. For such a CQ_F query, we define its *canonical graph* as its triple graph, to which we add an edge $\{x, y\}$ for each filter constraint that uses the two variables x and y. For all other CQ_F queries, its *canonical hypergraph* is obtained from its triple graph, to which we add a hyperedge $\{x_1, \ldots, x_k\}$ for each filter constraint that uses precisely the k variables $x_1, \ldots, x_k$.

In our corpus, 81.07% (90.28%) of the CQ_F patterns are suitable for graph analysis.

We now additionally consider Opt. Pérez et al. [40] showed that unrestricted use of Opt in SPARQL patterns makes query evaluation PSPACE-complete, which is significantly more complex than the NP-completeness of CQs or CQ_F queries. They discovered that patterns that satisfy an extra condition called *well-designedness* [40], can be evaluated more efficiently. Letelier et al. [30] show that, in the presence of projection, evaluation of well-designed patterns is Σ_2^P-complete.

Definition 5 A SPARQL pattern P using only the operators And, Filter, and Opt is *well-designed* if for every occurrence i of an Opt-pattern $(P_1 \text{ Opt } P_2)$ in P, the variables from vars(P_2)\vars(P_1) occur in P only inside i.[20]

In our corpus, 98.74% (98.18%) of the AOF patterns are well-designed (but do not necessarily have simple filters). Unfortunately, it is not yet sufficient for well-designed patterns to have a hypergraph of constant hypertreewidth for their evaluation to be tractable [8]. However, Barceló et al. show that this can be mended by an additional restriction called *bounded interface width*. We explain this notion by example and refer to [8] for details.

Example 2 The following patterns come from [40,30]:
P_1 = (((?A, name, ?N) Opt (?A, email, ?E))
Opt (?A, webPage, ?W))
and P_2 = ((?A, name, ?N)
Opt ((?A, email, ?E) Opt (?A, webPage, ?W)))
Figure 4 has tree representations T_1 and T_2 for P_1 and P_2, respectively, called *pattern trees*. The pattern trees T_i are obtained from the parse trees of P_i by applying a standard encoding based on Currying [33, Section 4.1.1]. The encoding only affects the arguments of the Opt operators in the queries. If the query also uses And, then it should first be brought in *Opt-normal form* [40] and then turned into a pattern tree. The resulting pattern trees will then have a CQ in each of its nodes.

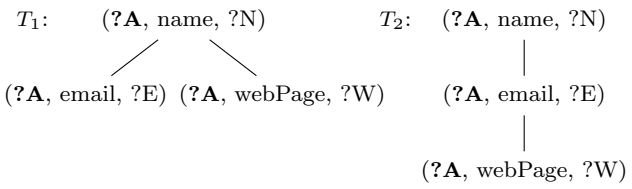


Fig. 4 Pattern trees that correspond to the queries in Example 2

Barceló et al. define pattern trees to be *well-designed* if, for each variable, the set of nodes in which it occurs forms a connected set. Notice that this is the case for T_1 and T_2. It would be violated in T_1 if the root would not use the variable ?A. Likewise, it would be violated in T_2 if the node labeled (?A, email, ?E) would not use the variable ?A.

The *interface width* of the pattern trees is the maximum number of common variables between a node and its child. Both trees in Figure 4 (and both queries P_1 and P_2) therefore have interface width one. (Common variables are bold in Figure 4.) If T_1 would use variable $?W$ instead of $?N$, then its interface width would be two.

Definition 6 A SPARQL pattern P using only the operators And, Filter, and Opt is a *CQ_{OF} query* if it has a well-designed pattern tree with interface width 1. It is *suitable for graph analysis* if it is a graph pattern and all its filter conditions are simple. The canonical graph and hypergraph of a CQ_OF query is defined analogously to that of CQ_F queries. That is, its triple graph (resp. hypergraph) is augmented with edges $\{x, y\}$ for each filter constraint that uses precisely the variables x and y (resp. hyperedges $\{x_1, \ldots, x_k\}$ for each filter constraint that uses precisely the variables $x_1, \ldots, x_k$).

We discovered that 98.72% (98.13%) of the AOF patterns are CQ_OF queries, which is almost equal to the number of well-designed patterns. Moreover, 85.30% (93.87%) are CQ_OF patterns that are suitable for graph analysis.

The Values keyword was used in 2.24% (5.61%) of the queries. It is particularly often used in `DBpedia17`, where it appears in 4.03% (13.37%) of the queries. The purpose of Values blocks is to test if a variable (or a tuple of variables) appears in a set that is given in the query. For instance, the subquery

```
VALUES (?country) {"Belgium" "France" "Germany"}
```

[20] Perez et al.'s definition also has a safety condition on the filter statements of the patterns, but the omission of this condition does not affect the results in this paper.

restricts the variable `?country` to be assigned to one of the values `"Belgium"`, `"France"`, or `"Germany"`. The Values block is used almost exclusively for unary conditions, that is, to test if the value of a single variable is in a given set of constants. However, it can also be used to test higher arity constraints, as in the subquery

```
VALUES (?x ?y) {(:a :b) (:a :c)}
```

which imposes a *binary* constraint, i.e., it binds the variable pair `(?x ?y)` to one of the two pairs in the body of the Values block. Concerning our shape analysis, we distinguish between Values blocks that use constraints of arity two or less and the others.

Definition 7 A *CQ_{OFV} query* is a SPARQL pattern P using only the operators And, Filter, Opt, and Values, such that the pattern obtained from P by removing all Values blocks is a CQ_{OF} query. It is *suitable for graph analysis* if all filters are simple and all values blocks have arity at most two. If a CQ_{OFV} query is suitable for graph analysis, its *canonical graph* is obtained from the triple graph by augmenting it with an edge for each binary filter constraint, and an edge for each binary Values block. For every other CQ_{OFV} queries, its *canonical hypergraph* is obtained from the triple hypergraph by augmenting it with a hyperedge $\{x_1, \ldots, x_k\}$ for each filter- or values block that uses precisely the variables $x_1, \ldots, x_k$.

Table 4 Well-designedness (*wd*), weak well-designedness (*wwd*) and unions thereof

Property	AbsoluteV	RelativeV	AbsoluteU	RelativeU
wd	198,109,323	98.74%	63,677,171	98.18%
wwd	200,064,814	99.71%	64,749,468	99.83%
AOF	200,641,891	100.00%	64,857,889	100.00%
uwd	208,672,931	74.35%	69,279,286	88.72%
uwwd	210,638,343	75.05%	70,360,134	90.10%
AOFU	280,672,732	100.00%	78,088,794	100.00%

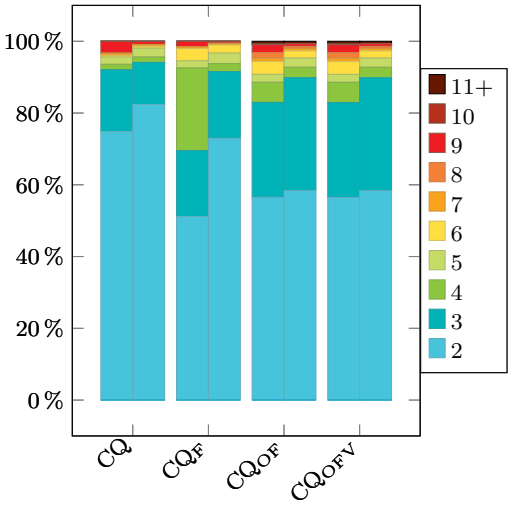


Fig. 5 Size of Valid (versus Unique) CQ-like queries with at least two triples.

5.4 (Weak) Well-Designedness And Unions

We conclude the section with a brief note on the usage of well-designedness with respect to the entire corpus of queries. Kaminski and Kostylev [27] defined a weaker version of well-designedness that has similar favorable computational properties. We therefore also analysed whether queries are *weakly well-designed*. Table 4 shows the number of AOF queries and the percentages thereof that are well-designed (*wd*) and weakly well-designed (*wwd*). We also took the set of queries that only use And, Opt, Filter, and Union (AOFU in Table 4) and investigated the percentages of queries thereof that are *unions* of wd or wwd queries. In most cases where the query is not a union of wd or wwd queries, it is because the union is not the top-level operator.

6 Shape Classification

In this section, we analyze the shapes of the canonical graphs and the tree- and hypertree width of CQ, CQ_{F}, CQ_{OF}, and CQ_{OFV} queries. We start with a note on the size of these queries. Figure 5 shows the respective sizes of these queries that have at least two triples by considering both Valid and Unique queries side by side. The fractions of queries with one triple are 90.65% (85.36%), 87.71% (83.22%), 81.54% (76.99%) and 81.81%(77.81%) for CQ, CQ_{F}, CQ_{OF} and CQ_{OFV} respectively. Unsurprisingly, small queries are more likely to be in one of these fragments and, therefore, simple queries are represented even more in these data sets than in the overall data set. Nevertheless, we have CQs and CQ_{F} queries with up to 81 triples and CQ_{OF} and CQ_{OFV} queries with up to 211 triples.

6.1 Graph Structure

We analyse the graph structure of queries. We only consider graphs for queries that were defined to be *suitable for graph analysis* in Section 5.1. We consider the remaining 27.27 million queries in CQ_{OF} in Section 6.2.

We first recall or define the basic shapes of the canonical graphs that we will study in this section. The shapes *chains* and *cycle* are already defined in Section 5.2. A *chain set* is a graph in which every connected component is a chain. (So, each chain is also a chain set.)

Table 5 Cumulative shape analysis of graph patterns in CQ, CQ_F, CQ_{OF}, and CQ_{OFV}, across all logs. The relative numbers are w.r.t. the queries that are suitable for graph analysis.

VALID	CQ/graph		CQ_F/graph		CQ_{OF}/graph		CQ_{OFV}/graph	
Shape	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*
no edge	73,147	0.06%	73,155	0.05%	73,155	0.04%	74,891	0.04%
$\leq$ 1 edge	107,268,916	90.71%	137,634,760	87.56%	139,234,499	81.35%	141,642,411	81.49%
chain	116,816,836	98.78%	151,963,617	96.68%	159,787,714	93.36%	162,216,710	93.32%
star	117,683,253	99.52%	155,325,069	98.82%	168,220,691	98.29%	170,671,088	98.19%
tree	118,059,399	99.83%	155,716,314	99.07%	168,936,241	98.71%	171,386,859	98.60%
flower	118,225,680	99.98%	156,730,621	99.71%	170,174,607	99.43%	172,922,659	99.48%
chain set	116,835,460	98.80%	151,990,203	96.70%	159,931,312	93.45%	162,287,197	93.36%
forest	118,078,726	99.85%	155,748,689	99.09%	169,089,411	98.80%	171,466,918	98.64%
bouquet	118,245,059	99.99%	156,763,406	99.73%	170,328,206	99.52%	173,003,151	99.53%
tw $\leq$ 2	118,254,672	100.00%	157,183,767	100.00%	171,147,726	100.00%	173,822,690	100.00%
tw $\leq$ 3	118,254,676	100.00%	157,183,771	100.00%	171,147,730	100.00%	173,822,694	100.00%
total	118,254,676	100.00%	157,183,771	100.00%	171,147,730	100.00%	173,822,694	100.00%

UNIQUE	CQ/graph		CQ_F/graph		CQ_{OF}/graph		CQ_{OFV}/graph	
Shape	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*
no edge	1,279	0.00%	1,284	0.00%	1,284	0.00%	1,661	0.00%
$\leq$ 1 edge	31,785,575	85.41%	46,480,574	83.05%	46,772,128	76.82%	48,866,909	77.37%
chain	36,839,344	98.99%	54,131,560	96.72%	56,042,768	92.05%	58,142,029	92.06%
star	37,123,785	99.75%	55,417,051	99.02%	60,203,786	98.89%	62,306,677	98.65%
tree	37,184,810	99.92%	55,487,815	99.15%	60,311,400	99.06%	62,414,439	98.82%
flower	37,202,015	99.96%	55,892,860	99.87%	60,735,713	99.76%	63,010,697	99.77%
chain set	36,851,176	99.02%	54,150,770	96.76%	56,096,837	92.14%	58,196,121	92.15%
forest	37,197,115	99.95%	55,509,443	99.19%	60,370,204	99.16%	62,473,266	98.92%
bouquet	37,214,357	100.00%	55,914,792	99.91%	60,794,835	99.86%	63,069,846	99.86%
tw $\leq$ 2	37,216,150	100.00%	55,965,143	100.00%	60,881,508	100.00%	63,156,533	100.00%
tw $\leq$ 3	37,216,153	100.00%	55,965,146	100.00%	60,881,511	100.00%	63,156,536	100.00%
total	37,216,153	100.00%	55,965,146	100.00%	60,881,511	100.00%	63,156,536	100.00%

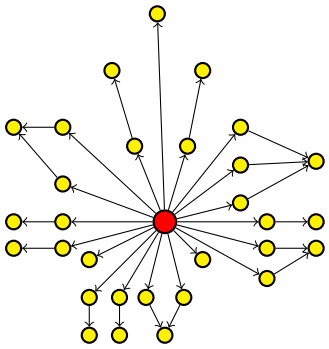


Fig. 6 An example of a flower query found in our DBpedia query logs (we added arrows to indicate the edge directions in the query; labels are omitted).

A *tree* is an undirected graph such that, for every pair of nodes x and y, there exists exactly one undirected path from x to y. (Hence, every chain is also a tree.) A *forest* is a graph in which every connected component is a tree.

A *star* is a tree for which there exists at most one node with more than two neighbors, that is, there is at most one node u such that there exist u_1, u_2, and u_3, all pairwise different and different from u, for which $\{u, u_i\} \in E$ for each $i = 1, 2, 3$.

Inspired by the results obtained with gMark on synthetic queries, we proceeded with the analysis of the query logs by looking at the encountered query shapes. Here, we consider queries as edge-labeled graphs, as defined in Section 5. In the next subsection we also investigate the hypergraph structure.

We investigate CQs, CQ_F queries, CQ_{OF} queries, and CQ_{OFV} queries. The last three fragments are interesting in that they bring under scrutiny more queries than the plain CQ set of query logs (by an increase of roughly 33% (50%), 44% (64%), and 47% (70%) respectively). We first wanted to identify classical query shapes, such as all variants of tree-like shapes (single edges, chains, sets of chains, stars, trees, and forests). The results are summarized in Table 5. From the analysis, we can draw the following observations. While tree-shaped queries even in their simple forms (chain of length 1 or single edges) are very frequent, the only observed exception occurs with star queries, which have

Table 6 Cumulative shape analysis of graph patterns in CQ, CQ_F, CQ_{OF}, and CQ_{OFV}, *after removal of IRIs*, across all logs. The relative numbers are w.r.t. the queries that are suitable for graph analysis.

VALID	CQ/graph		CQ_F/graph		CQ_{OF}/graph		CQ_{OFV}/graph	
Shape	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*
no edge	106,952,766	90.44%	136,357,792	86.75%	144,549,634	84.46%	144,643,932	83.21%
≤ 1 edge	116,643,820	98.64%	150,954,951	96.04%	160,737,562	93.92%	163,386,730	94.00%
chain	117,774,655	99.59%	155,832,073	99.14%	167,819,465	98.06%	170,472,931	98.07%
star	117,876,831	99.68%	156,787,151	99.75%	170,062,935	99.37%	172,737,353	99.38%
tree	118,235,060	99.98%	157,146,906	99.98%	170,423,705	99.58%	173,098,147	99.58%
flower	118,243,330	99.99%	157,162,189	99.99%	170,439,245	99.59%	173,114,179	99.59%
chain set	117,785,058	99.60%	155,852,116	99.15%	167,851,157	98.07%	170,504,640	98.09%
forest	118,245,559	99.99%	157,167,354	99.99%	170,732,618	99.76%	173,407,077	99.76%
bouquet	118,253,840	100.00%	157,182,660	100.00%	170,748,181	99.77%	173,423,132	99.77%
tw ≤ 2	118,254,674	100.00%	157,183,769	100.00%	171,147,728	100.00%	173,822,692	100.00%
tw ≤ 3	118,254,676	100.00%	157,183,771	100.00%	171,147,730	100.00%	173,822,694	100.00%
total	118,254,676	100.00%	157,183,771	100.00%	171,147,730	100.00%	173,822,694	100.00%

UNIQUE	CQ/graph		CQ_F/graph		CQ_{OF}/graph		CQ_{OFV}/graph	
Shape	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*
no edge	32,886,654	88.37%	47,048,004	84.07%	49,453,297	81.23%	49,490,285	78.36%
≤ 1 edge	36,511,703	98.11%	52,939,383	94.59%	56,293,258	92.46%	58,562,031	92.73%
chain	37,150,107	99.82%	55,596,772	99.34%	60,134,203	98.77%	62,405,465	98.81%
star	37,164,017	99.86%	55,898,107	99.88%	60,743,607	99.77%	63,018,260	99.78%
tree	37,210,340	99.98%	55,944,891	99.96%	60,791,019	99.85%	63,065,687	99.86%
flower	37,213,881	99.99%	55,954,879	99.98%	60,801,134	99.87%	63,076,131	99.87%
chain set	37,151,726	99.83%	55,605,967	99.36%	60,148,326	98.80%	62,419,603	98.83%
forest	37,212,024	99.99%	55,954,365	99.98%	60,834,660	99.92%	63,109,343	99.93%
bouquet	37,215,574	100.00%	55,964,373	100.00%	60,844,795	99.94%	63,119,807	99.94%
tw ≤ 2	37,216,152	100.00%	55,965,145	100.00%	60,881,510	100.00%	63,156,535	100.00%
tw ≤ 3	37,216,153	100.00%	55,965,146	100.00%	60,881,511	100.00%	63,156,536	100.00%
total	37,216,153	100.00%	55,965,146	100.00%	60,881,511	100.00%	63,156,536	100.00%

very low occurrence with respect to the other tree-like shapes.

Since simple queries are overrepresented in query logs (already over 87.76% (83.23%) of CQ_F patterns uses only one triple, for example), it is no surprise that the overwhelming majority of the queries is acyclic, i.e., a forest. However, we also wanted to get a better understanding of the more *complex* queries in the logs, so we also investigated the cyclic queries. Our goal is to obtain a cumulative shape analysis where simpler shapes are subsumed by more sophisticated query shapes, with the latter reaching almost 100% coverage of the query logs.

A first observation was that plain cycles are not very common. By visually inspecting the remaining cyclic queries, we observed that many of them could be seen as a node with simple attachments, which we call *flower*.

Definition 8 A *petal* is a graph consisting of a source node s, target node t, and a set of at least two node-disjoint paths from s to t. (For instance, a cycle is a petal that uses two paths.) A *flower* is a graph consisting of a node x with three types of attachments: chains (the *stamens*), trees that are not chains (the *stems*), and *petals*. As an edge case, we also consider the empty graph to be a flower.

An example of a real flower query posed by users in one of our DBpedia logs is illustrated in Figure 6. It consists of a central node with four petals (one of which using three paths), ten stamens and zero stems attached.

We also considered sets of flowers, which we called *bouquets*, to further increase the ratio of queries that could be classified from the original logs. The number of flowers and bouquets in the query logs only overcome those of trees and forests by roughly 0.01%–0.09% (0.03–0.10%) for all the four fragments. Furthermore, for all fragments, the majority of the cyclic queries is captured by bouquets.

In the above analysis, we have analyzed the shapes of queries when the latter are represented as graphs as defined in Section 5, i.e., the nodes can be either variables or constants. Constants are in fact helpful for us to obtain a rough idea of the shape of patterns that users try to find in graphs, but research on query optimization often focuses on the shape of patterns *without*

constants. (The reason is that constants can typically be matched to only one node in the graph and therefore do not highly contribute to the complexity of evaluation.) For that reason, we have rerun the above analysis on queries excluding constants in order to identify the differences in the obtained shape classification. The most significant observation here is that many shapes disintegrate to a set of variables (i.e., no more edges are present in their graph). More precisely, for the four fragments CQ, CQ_F, CQ_{OF}, and CQ_{OFV}, we have that respectively 90.44% (88.37%), 86.75% (84.07%), 84.46 (81.23%), and 83.21% (78.36%) of the queries that are suitable for graph analysis have no more edges when considering the restriction of their canonical graphs to variables only. This is a huge change, since such shapes only constituted 0.00%–0.06% of the shapes of queries with constants in Table 5.

As a final remark, we can notice that the shift from shapes with constants to shapes with only variables is significantly affecting the "no edge" fragment and has less impact on the other shapes. For the "no edge" fragment, many queries boil down to a set of isolated nodes or to a singleton when constants are removed. We could not observe in both Tables huge differences between the Valid and Unique query logs, that rather resemble each other in terms of relative percentages of shapes.

6.2 Tree- and Hypertreewidth

It is well-known that the tree- or hypertreewidth of queries are important indicators to gauge the complexity of their evaluation. We therefore investigated the tree- and hypertreewidth of CQ, CQ_F CQ_{OF}, and CQ_{OFV} queries. We do not formally define tree- or hypertreewidth in this paper but instead refer to an excellent introduction [19]. In the terminology of Gottlob et al., we investigate the *treewidth* of the graphs of the queries and the *generalized hypertree width* of the canonical hypergraphs of queries.

Treewidth. All shapes we discussed in Section 6.1 have treewidth at most two. Forests (and all subclasses thereof) have treewidth one, whereas flowers and bouquets have treewidth two. We investigated the remaining queries using the tool[21] JDrasil [7] and discovered that three queries had treewidth three (one such query is in Figure 7) and all others had treewidth two, see Table 5. This new tool let us compute the treewidth of the queries in our corpus, whereas in the conference version of the paper we used `detkdecomp`, which outputs the generalized hypertreewidth. The latter can be lower than the treewidth, thus the results reported here exhibit more precision. From the treewidth perspective, it is interesting to note that many queries of treewidth two are also *flowers* or *bouquets* (Definition 8), which are a very restricted fragment.

Hypertree Width. We recall that we only considered the graph of queries for which variables in the predicate position are not re-used elsewhere (if they occur at all). In CQ_{OFV}, 58,782,592 (17,333,741) queries used a variable in a predicate condition or a filter or values condition of arity more than two, and we therefore considered their hypergraph structure, without constants, to assess the cyclicity of these queries. We determined their generalized hypertree width with the tool `detkdecomp` from the Hypertree Decompositions home page [17]. Furthermore, we measure the cyclicity of the hypergraphs *without constants*, as it is usually done in the literature.

Our results are summarized in Table 7, which contains the hypertreewidth of queries from CQ, CQ_F, CQ_{OF}, and CQ_{OFV} that were not yet analysed in Section 6.1. Concerning CQs, all the remaining queries had hypertree width one, except for 68 (56) queries with hypertree width two and eight queries with hypertree width three. In the largest fragment, CQ_{OFV}, we have 542,409 (242,941) such queries with hypertreewidth two and nine with hypertreewidth three. So, especially in the fragment CQ_{OFV}, we see a significant portion of the queries that exhibits cyclicity, i.e., 8.03% of the unique queries. This means that considering Values constructs indeed can have an impact on the cyclicity of queries.

We also looked at the number of nodes in the hypertree decompositions that the tool gave us, since this number can be a guide for how well *caching* can be exploited for query evaluation [26] (the higher the number, the better caching can be exploited). For the queries with hypertree width one, the number of nodes in the decompositions corresponds to their number of edges, which can already be seen in Figure 5. (Nevertheless, we found several hundred queries in CQ_{OFV} queries with 100 or more nodes in their hypertree decompositions, the vast majority occurring in the DBpedia logs.) Finally, out of the queries with hypertreewidth two, 598 (465) had decompositions of size more than 10, going up to a maximum of 16. The CQ_{OFV} queries of hypertreewidth three all had decompositions of size smaller than 10, except for one query in `DBpedia17` which had a decomposition of size 33.

7 Analysis of the Shapes

In this section, we provide a deeper characterization of the query shapes found in our large corpus, by present-

[21] Available on `https://maxbannach.github.io/Jdrasil/`

Table 7 Hypertreewith (*htw*) of the queries that were not analysed in Section 6.1, i.e., queries that use filter- or values conditions of arity three or more; or that re-use some variable in the predicate position elsewhere

	CQ				CQ_F			
	AbsoluteV	RelativeV	AbsoluteU	RelativeU	AbsoluteV	RelativeV	AbsoluteU	RelativeU
$htw = 1$	4,137,042	100.00%	2,338,797	100.00%	5,162,377	95.42%	2,557,651	99.17%
$htw = 2$	68	0.00%	56	0.00%	248,050	4.58%	21,410	0.83%
$htw = 3$	8	0.00%	8	0.00%	8	0.00%	8	0.00%
Total new	4,137,118	100.00%	2,338,861	100.00%	5,410,435	100.00%	2,579,069	100.00%

	CQ_{OF}				CQ_{OFV}			
	AbsoluteV	RelativeV	AbsoluteU	RelativeU	AbsoluteV	RelativeV	AbsoluteU	RelativeU
$htw = 1$	26,680,385	99.07%	2,743,833	99.22%	26,725,649	98.01%	2,780,838	91.97%
$htw = 2$	249,126	0.93%	21,678	0.78%	542,409	1.99%	242,941	8.03%
$htw = 3$	8	0.00%	8	0.00%	9	0.00%	9	0.00%
Total new	26,929,519	100.00%	2,765,519	100.00%	27,268,067	100.00%	3,023,788	100.00%

Table 8 Analysis of longest paths in chain, star, and tree queries (*Valid* and *Unique* queries)

longest path length	#V chain	Relative %	#V star	Relative %	#V tree	Relative %
1	142,644,649	87.34%				
2	16,185,787	9.91%	7,884,906	92.42%		
3	3,880,284	2.38%	376,217	4.41%	59,537	8.00%
4	601,580	0.37%	264,287	3.10%	284,953	38.29%
5	1,970	0.00%	6,408	0.08%	14,167	1.90%
6	2,132	0.00%	136	0.00%	385,110	51.75%
7	1,011	0.00%	10	0.00%	436	0.06%
8	1,015	0.00%	8	0.00%	2	0.00%
9	4	0.00%	7	0.00%	0	0.00%
10–23	8	0.00%	11	0.00%	2	0.00%
total	163,318,440	100.00%	8,531,990	100.00%	744,207	100.00%
longest path length	#U chain	Relative %	#U star	Relative %	#U tree	Relative %
1	49,039,098	84.01%				
2	6,853,199	11.74%	3,833,545	91.21%		
3	2,400,853	4.11%	212,739	5.06%	17,213	15.56%
4	76,828	0.13%	155,883	3.71%	31,779	28.73%
5	1,333	0.00%	901	0.02%	12,752	11.53%
6	1,468	0.00%	50	0.00%	48,792	44.11%
7	1,009	0.00%	8	0.00%	79	0.07%
8	1,011	0.00%	8	0.00%	2	0.00%
9	3	0.00%	6	0.00%	0	0.00%
10–23	7	0.00%	7	0.00%	2	0.00%
total	58,374,809	100.00%	4,203,147	100.00%	110,619	100.00%

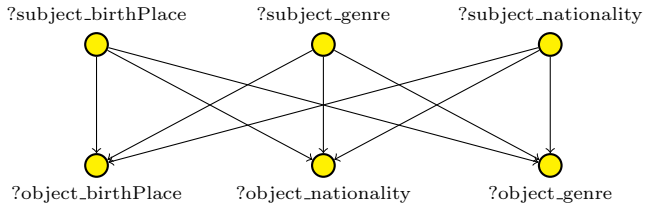


Fig. 7 The DBpedia query exhibiting tree width equal to 3

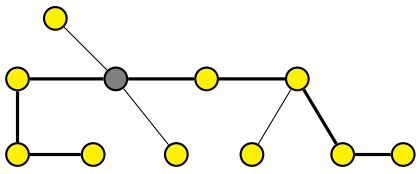


Fig. 8 A tree-shaped query with longest path of length 7 (in bold) and maximal degree of nodes equal to 4 (for the grey node).

ing various measures of these shapes. We first focus on chain, tree, and star-shaped queries, which are the most recurrent shapes in our logs and we identify some measures for the ensemble of these shapes or separately for each class. At the end of the Section, we also provide more insights about the cyclic queries found in our logs.

Table 9 Maximal degree of nodes in star and tree queries (*Valid* and *Unique*)

max degree	#V star	Relative %	#U star	Relative %	#V tree	Relative %	#U tree	Relative %
3	5,791,971	67.89%	3,173,041	75.49%	401,873	54.00%	73,125	66.11%
4	1,183,578	13.87%	406,272	9.67%	26,154	3.51%	2,640	2.39%
5	350,676	4.11%	191,479	4.56%	279,092	37.50%	30,844	27.88%
6	710,511	8.33%	228,573	5.44%	31,258	4.20%	3,305	2.99%
7	223,651	2.62%	68,179	1.62%	5,367	0.72%	589	0.53%
8	78,890	0.92%	55,056	1.31%	375	0.05%	51	0.05%
9	38,711	0.45%	25,152	0.60%	47	0.01%	36	0.03%
10–19	147,266	1.73%	53,067	1.26%	39	0.01%	27	0.02%
20–29	2,758	0.03%	2,077	0.05%	2	0.00%	2	0.00%
30–39	230	0.00%	192	0.00%				
40–49	64	0.00%	51	0.00%				
50–59	6	0.00%	6	0.00%				
60–63	3,678	0.04%	2	0.00%				
total	8,531,990	100.00%	4,203,147	100.00%	744,207	100.00%	110,619	100.00%

Table 10 Number of high-degree nodes (#HD) in tree shaped queries (*Valid* and *Unique*)

#HD	#V tree	Relative %	#U tree	Relative %
2	59,537	8.00%	17,213	15.56%
3	281,184	37.78%	31,197	28.20%
4	14,348	1.93%	12,877	11.64%
5	365,318	49.09%	47,920	43.32%
6	23,811	3.20%	1,405	1.27%
7	7	0.00%	5	0.00%
9	1	0.00%	1	0.00%
11	1	0.00%	1	0.00%
total	744,207	100.00%	110,619	100.00%

Table 11 Average degree of inner nodes (AvgDeg) in tree shaped queries (*Valid* and *Unique*)

AvgDeg	#V tree	Relative %	#U tree	Relative %
2–2.9	400,426	53.81%	61,955	56.01%
3–3.9	308,649	41.47%	44,358	40.10%
4–4.9	34,514	4.64%	4,027	3.64%
5–5.9	346	0.05%	160	0.14%
6–6.9	103	0.01%	52	0.05%
7–7.9	157	0.02%	58	0.05%
8–8.9	12	0.00%	9	0.01%
total	744,207	100.00%	110,619	100.00%

An immediate measure of the span of a query shape is the size of the longest (undirected) path in the query. Such a measure is readily applicable to chains, stars and tree-shaped queries. The size of the longest path for a tree-shaped query is the length of the longest path from one leaf to another leaf. For instance, if we consider the tree-shaped query in Figure 8, we observe that its longest path has length 7 (highlighted in bold). The same applies to star-shaped queries where the longest path is the path from one vertex to another traversing the central node of the star, whereas the longest path in a chain is the length of the chain itself.

Table 12 Maximal cycle length in cyclic queries

MaxCyc	# Valid	# Unique
3	1,455,724	328,118
4	51,308	23,946
5	25,062	5,865
6	3,243	79
7	7	7
8	1	1
10	1	1
total	1,535,346	358,017

Table 13 Minimal cycle length in cyclic queries

MinCyc	# Valid	# Unique
3	1,456,037	328,347
4	51,023	23,739
5	25,048	5,853
6	3,230	70
7	7	7
10	1	1
total	1,535,346	358,017

Table 8 reports the lengths of the longest paths in chain, tree, and star-shaped queries in our logs. We can notice that the longest paths in chain and star queries are majorly small (significant percentages go up to size of the longest path equal to 3 for chain queries and to 4 for star queries, respectively), whereas trees are somehow different. Their non-zero percentages characterize lengths of longest paths up to 6 for tree-shaped queries. In all shapes, we could find some examples of queries with quite long paths (from length 10 to 23) and these are comparably higher in chains and stars than in tree-shaped queries.

We then proceeded with the analysis of the shapes by focusing on the nodes with the maximal degree of nodes in star- and tree-shaped queries. In our example of a tree-shaped query in Figure 8, we can easily see that

Table 14 Free-connex acyclicity (FCA) and *htw* of all the CQs in our logs.

	CQ				CQ_F			
	AbsoluteV	RelativeV	AbsoluteU	RelativeU	AbsoluteV	RelativeV	AbsoluteU	RelativeU
FCA	117,669,790	96.14%	36,786,611	93.00%	152,870,355	93.98%	53,393,254	91.19%
$htw \leq 1$	118,245,559	96.61%	37,212,024	94.08%	157,167,354	96.63%	55,954,365	95.56%
$htw \leq 2$	122,391,781	100.00%	39,555,004	100,00%	162,654,843	100.00%	58,554,583	100.00%
$htw \leq 3$	122,391,794	100.00%	39,555,014	100.00%	162,654,856	100.00%	58,554,593	100.00%
Total	122,391,794	100.00%	39,555,014	100.00%	162,654,856	100.00%	58,554,593	100.00%
	CQ_{OF}				CQ_{OFV}			
	AbsoluteV	RelativeV	AbsoluteU	RelativeU	AbsoluteV	RelativeV	AbsoluteU	RelativeU
FCA	160,545,014	80.02%	55,059,069	84.89%	163,203,235	58.15%	57,331,127	73.42%
$htw \leq 1$	170,732,618	85,09%	55,954,365	86.27%	173,407,077	61.78%	63,109,343	80.82%
$htw \leq 2$	200,641,878	100.00%	64,857,879	100.00%	280,672,718	100.00%	78,088,783	100.00%
$htw \leq 3$	200,641,891	100.00%	64,857,889	100.00%	280,672,732	100.00%	78,088,794	100.00%
Total	200,641,891	100.00%	64,857,889	100.00%	280,672,732	100.00%	78,088,794	100.00%

the maximal degree of nodes is equal to 4. Obviously, this measure is not informative for chain queries, which are completely characterized by their length (and whose vertices have a maximal degree of two). Table 9 shows the results for stars and tree-shaped queries. The higher percentages of star queries have maximal degree of their vertices equal to 3, whereas for tree-shaped queries, the majority has maximal degree equal to 3 or 5. The highest values of maximal degrees can be observed in stars more than in tree-shaped queries.

We then focused on tree-shaped queries and computed the number of nodes we found with high degrees. This measure is only applicable to tree-shaped queries and neither to stars (that always have one node with highest degree) nor to chains. The results are shown in Table 10, where we can notice 49.09% (43.32%) of the tree-shaped queries have 5 high-degree vertices. We also found one query with 11 high-degree vertices.

We did not dig further into the actual values of the degrees for these high-degree nodes, even though a combined view of Table 8 and Table 9 provides a quick grasp on that.

Further investigating the tree shapes, we computed in Table 11 the average degrees of inner nodes in these shapes (again not applicable to chains and stars). We can observe that the majority of inner nodes degrees stay in between 2 and 4 on average.

Finally, we looked at the class of cyclic queries and measured the maximal and minimal cycle lengths of the cycles. The cycle computation considered again the queries as undirected graphs and aimed at constructing the cycle basis for such graphs. A cycle basis is formed from any spanning tree or spanning forest of the given graph, by selecting the cycles obtained by combining a path in the tree with a single edge outside the tree. In order to keep the computation of cycle basis polynomial, we set up an empirical bound (equal to 8) to the number of cycles that form the cycle basis. We thus counted the minimal and maximal cycle length of the discovered cycle basis of each query. Tables 12 and 13 report the results of this analysis for CQ_{OFV} queries.

We also computed the property of free-connex acyclicity for CQ, CQ_F, CQ_{OF} and CQ_{OFV}. A conjunctive query is free-connex acyclic if it is acyclic and the set of its free variables [22] is a connex subset of the join tree of the query [6]. The join tree of a query corresponds to the tree-structure of the acyclic hypergraph underlying the query. Free-connex acyclicity is interesting because it characterizes the conjunctive queries for which certain kinds of efficient algorithms exist for enumerating their output [6,24] (under standard complexity-theoretical assumptions). Table 14 shows the results by comparing the number of all conjunctive queries (including those that are not suitable for graph analysis and thus are not considered in Table 5) and the number of free-connex acyclic queries found in our logs. We can notice that the latter are abundant in all the fragments CQ, CQ_F, CQ_{OF} and CQ_{OFV}. For a cross comparison, we also show the hypertreewidth of all the conjunctive queries in our logs (and not only those reported in Table 5). We can observe that all the CQs in our logs have *htw* less or equal to 3.

8 Tree Pattern Queries

Tree pattern queries (e.g., [35,28,16,15]) are a well-studied query formalism on trees which is inspired on

[22] The free or distinguished variables of a query considered as a first-order propositional formula are the set of variables used as output in the formula.

XPath but which can just as well be used for querying graph-structured data [31,15].We next define a *tree-pattern-like* fragment of our queries and investigate how common it appears in the logs.

Property paths have the power to do *forward* and *backward* navigation through edges. For instance, if a is an IRI, then the property path $\hat{}a$ allows to follow an a-edge in the graph in backward direction. In the following definition, we only allow forward navigation. A *directed tree* is a connected, directed graph such that there is a unique node without incoming edges (the root) and, for all edges (u, v) and (u', v), we have that $u = u'$ (every node has at most one parent).

Definition 9 A *conjunctive regular path query (CRPQ)* is a SPARQL pattern that only uses triple patterns, the operator And, and property paths.

The *directed canonical graph of a CRPQ* P is the directed graph obtained from the edges $E \cup E_p$, where $E = \{(x, y) \mid (x, \ell, y)$ is a triple pattern in P and $\ell \in \mathcal{I} \cup \mathcal{V}\}$ and $E_p = \{(x, y) \mid (x, pp, y)$ is a property path pattern in $P\}$.

Definition 10 A CRPQ P is a *tree pattern query* if

- its directed canonical graph is a directed tree and
- every property path is a concatenation of IRIs and property paths of the form a^*, where a is an IRI.

Our analysis shows that 99.77% (99.91%) of the CRPQs have a canonical graph that is an *undirected* tree. Out of these, 87.92% (84.96%) are tree pattern queries. This is a fairly significant number, considering that we require the shape to be a *directed* tree. If we additionally allow the Filter operator (in a similar way as in Section 6), these percentages remain roughly the same.

9 Property Paths

We found 1,412,762 (329,984) queries using property paths in our corpus. From these queries, we extracted 1,528,701 (404,721) property paths in total, which is about 67% more than the 247,404 unique property paths considered in [11]. Although property paths are therefore rare in relation to the entire corpus, this is not so for every data set: 92 queries (29.87%) in `Wikidata17` have property paths.[23]

A large fraction of these property paths are extremely simple. For instance, 65,693 (63,428) property paths are $!a$ ("follow an edge not labeled a") and 80,421 (58,156) are $\hat{}a$ ("follow an a-edge in reverse direction"). In total, 65,751 (63,478) queries use the *different-from* operator "!" and 394,726 (144,569) use the reverse navigation operator "^".

In Table 15, we present an overview of all the property paths we found in the corpus. For readability, we don't explicitly denote the concatenation operator "/", so we write ab instead of a/b. In our classification, we treat $\hat{}a$ and $!a$ the same as a literal. For instance, we classify ab, $(\hat{}a)b$, and $(!a)b$ all as $a_1 \cdots a_k$ with $k = 2$. We use capital letters to denote subexpressions that can match a set of different IRIs. For example, $(a|b)$ can match a and b, i.e., a set of two symbols. In the column *Set Sizes*, we wrote these sizes of sets we found. If the expression uses the !-operator, it can actually be matched by an infinite number of IRIs and can be seen as a wildcard test. (Some users even write the expression $(!a|!b)$ to obtain a wildcard that can match any IRI.) If we found expressions that use the !-operator, we annotate this with (wc) in the *Set Sizes* column.

Furthermore, each row represents the expression type listed on the left plus its symmetric form. For instance, when we write a^*b, we count the expressions of the form a^*b and ba^*. The variant listed in the table is the one that occurred most often in the data. That is, a^*b occurred more often than ba^*.

In the new corpus, we could enumerate a total of 111 different property paths, regrouped into 35 classes. This corresponds to an increase of roughly one third in the number of different property paths and classes (respectively equal to 87 and 22 in the conference version of this paper [11].) The occurrences of classes already found in [11] is roughly preserved in the new corpus if we focus on Unique queries. However, the new analysis presented here includes the percentages of occurrences in the logs of Valid queries, which is interesting by itself. For instance, the transitive closure of a single label a^+ is quite prominent in the Valid queries (more than 40% compared to 2% in the logs of Unique queries in the previous corpus).

Bagan et al. [5] proved a dichotomy on the data complexity of evaluating property paths under a *simple path* semantics, i.e., expressions can only be matched on paths in the RDF graph in which nodes appear only once. They showed that, although evaluating property paths under this semantics is NP-complete in general, it is possible in PTIME if the expressions belong to a class called C_{tract}. Remarkably, we only found eight expressions in our corpus which are not in C_{tract}, namely $(ab)^*$ (once) and $ab(ab)^*$ (seven times). The complexity of enumerating answers to property paths of the form as in Table 15 is studied in [34]. More precisely, the paper investigates enumeration problems for *simple*

[23] Even though our set of Wikidata queries is very small, Malyshev et al. [32] recently found a similar percentage of property path usage in Wikidata logs consisting of $\sim$ 480M valid queries.

Table 15 Structure of property paths in our corpus. Capital letters denote unions of symbols or wildcards.

Expression Type	*Absolute V*	*Relative V*	*Absolute U*	*Relative U*	*Set Sizes*	*Values for k*
a^+	618,459	40.46%	5,968	1.47%		
A^*	361,402	23.64%	89,379	22.08%	≤ 4 (wc)	
a^*	160,628	10.51%	68,681	16.97%		
a^*b	23,523	1.54%	20,566	5.08%		
a^*b^*	14,674	0.96%	997	0.25%		
$A^*B?$	7,252	0.47%	1,326	0.33%	≤ 5	
abc^*	70	0.00%	54	0.01%		
$(ab^*)\|c$	45	0.00%	15	0.00%		
$a^*b?$	45	0.00%	15	0.00%		
A^+	19	0.00%	18	0.00%	≤ 7 (wc)	
$ab(ab)^*$	7	0.00%	7	0.00%		
$a^+\|b^+$	3	0.00%	3	0.00%		
Ab^*	2	0.00%	1	0.00%	≤ 1 (wc)	
aB^*	2	0.00%	2	0.00%	≤ 2 (wc)	
$a\|b^*$	2	0.00%	2	0.00%		
$a\|b^+$	2	0.00%	2	0.00%		
$A^+B?$	1	0.00%	1	0.00%	≤ 5	
A^*B	1	0.00%	1	0.00%	≤ 5	
A^*bc	1	0.00%	1	0.00%	$= 5$	
$a?b^*$	1	0.00%	1	0.00%		
$(ab)^*$	1	0.00%	1	0.00%		
A	139,662	9.14%	129,515	32.00%	≤ 6 (wc)	
$a_1 \cdots a_k$	109,166	7.14%	25,431	6.28%		≤ 6
$\hat{}a$	80,421	5.26%	58,156	14.37%		
$a?$	9,864	0.65%	3,347	0.83%		
$a_1? \cdots a_k?$	2,704	0.18%	971	0.24%		≤ 5
$a_1? \cdots a_{k-1}?a_k$	664	0.04%	197	0.05%		≤ 3
$aB?$	40	0.00%	34	0.01%	≤ 2	
$ab?c?d$	12	0.00%	10	0.00%		
Ab	8	0.00%	6	0.00%	≤ 2	
AB	7	0.00%	4	0.00%	≤ 2	
$a\|ba\|c\|d$	6	0.00%	2	0.00%		
$A?$	4	0.00%	4	0.00%	≤ 2 (wc)	
$abc?d?$	2	0.00%	2	0.00%		
$AAAAAA$	1	0.00%	1	0.00%	$= 2$	
Total	1,528,701	100%	404,721	100%		

transitive expressions, which capture 99.03% (99.74%) of the expressions in Table 15.

10 Evolution of Queries over Time

In a typical usage scenario of a SPARQL endpoint, a user queries the data and gradually refines her query until the desired result is obtained. In this section, we analyse to which extent such behavior occurs. The results are very preliminary but show that, in certain contexts, it can be interesting to investigate optimization techniques for sequences of similar queries.

We consider a query log to be an ordered list of queries $q_1, \ldots, q_n$. We introduce the notion of a *streak*, which intuitively captures a sequence of similar queries within close distance of each other. To this end we assume the existence of a *similarity test* between two queries. We then say that queries q_i and q_j with $i < j$ *match* if (1) q_i and q_j are similar and (2) no query $q_{i'}$ with $i < i' < j$ is similar to q_i. A *streak (with window size w)* is a sequence of queries $q_{i_1}, \ldots, q_{i_k}$ such that, for each $\ell = 1, \ldots, k-1$, we have that $i_{\ell+1} - i_\ell \leq w$ and $q_{i_{\ell+1}}$ matches q_{i_ℓ}.

In theory, it is possible for a query to belong to multiple streaks. E.g., it is possible that q_1 and q_2 do not match, but query q_3 is sufficiently similar to both. In this case, q_3 belongs to both streaks q_1, q_3 and q_2, q_3.

In the present study, we used Levenshtein distance as a similarity test. More precisely, we said that two queries are *similar* if their Levenshtein distance, after removal of namespace prefixes, is at most 25%.[24] We removed namespace prefixes prior to measuring their Levenshtein distance, because they introduce superficial similarity. As such, we require queries to be at least 75% identical starting from the first occurrence of the

[24] We normalized the measure by dividing the Levenshtein distance by the length of the longer string.

Table 16 Length of streaks in three single-day logs

Streak length	*#DBP'14*	*#DBP'15*	*#DBP'16*
1–10	42,272	167,292	199,375
11–20	3,732	24,001	37,402
21–30	2,425	4,813	17,749
31–40	884	667	5,849
41–50	283	162	1,998
51–60	88	40	711
61–70	27	8	322
71–80	15	4	129
81–90	5	1	47
91–100	5	0	27
>100	4	0	24

keywords Select, Ask, Construct, or Describe. We took a window size of 30.

Streak Length. Since the discovery of streaks was extremely resource-consuming, we only analysed streaks in three randomly selected log files from `DBpedia14`, `DBpedia15`, and `DBpedia16`. The sizes of these log files, each reflecting a single day of queries to the endpoint, were 273MiB, 803MiB, and 1004MiB respectively.

For the ordering of the queries, we simply considered the ordering in the log files, since the logs are sorted over time.

The results on streak length are in Table 16. Using window size 30, the longest streak we found had length 169 and was in the 2016 log file. When we increased the window size, we noticed that it was still possible to obtain longer streaks. We believe that a more refined analysis on the encountered streaks can be carried out when tuning the window size and deriving more complex metrics on the similarity of the queries within each streak. These issues are, however, subject of further research, which we plan to pursue in future work.

Evolution of Size and Structure. In addition to the length of streaks, we also investigated how the number of triples and structure of queries in streaks change over time. To this end, we needed to parse the queries in streaks. The three log files contain a combined amount of 510,361 streaks. Out of these streaks, 321,042 have at least two queries and 234,627 additionally have at least one query that parses. Remarkably, in the latter set, only 1,402 streaks have an erroneous query. Here, 1,202 have an erroneous query followed by a correct one, and 789 have a correct query followed by an erroneous one.

We then investigated the number of triples of queries in streaks. We have 355,466 streaks which have at least one parsable query that contains at least one triple.[25]

[25] For 88,201 streaks, all queries had an empty body. Another 31 streaks had a non-empty body, containing no triples.

Table 17 Largest query occurring in streaks

Max #Triples	*#Streaks*	*Max #Triples*	*#Streaks*
1	130,706	13–20	9,509
2	41,811	21–30	544
3	34,081	31–40	233
4	9,990	41–50	86
5	3,325	51–60	44
6	1,733	61–70	32
7	8,465	71–80	17
8	10,604	81–90	11
9	7,837	91–100	3
10	1,080	101–110	9
11	51,521	> 110	7
12	43,819		

Table 18 Structures of queries appearing in the same streak (*chn* = chain, *bt* = 'branching tree', i.e., tree that is not a chain, *cyc* = cyclic)

Shapes	*#Streaks*
containing *chn*	148,632
consisting only of *chn*	147,106
containing *bt*	39,839
consisting only of *bt*	39,810
containing *cyc*	526
consisting only of *cyc*	493
containing *bt* and *cyc*	12
consisting only of *bt* or *cyc*	40,315
containing *bt* and *chn*	2
consisting only of *bt* or *chn*	186,918
containing *chn* and *cyc*	21
consisting only of *chn* or *cyc*	147,620
consisting only of *chn*, *bt*, or *cyc*	187,444

Table 17 contains, for each of the 355,466 streaks, what is the maximal number of triples in any of its queries. We noticed that this number is quite stable: we only have 3,915 streaks in which this number changes during the streak.

Table 18 contains results on the shapes of queries in streaks. We considered chain queries, trees that branch (and therefore are no chains), and cyclic queries, that is, queries that contain a cycle. Table 18 contains, for each subset S of these three shapes, the number of streaks that *contain only shapes from* S and the number of streaks that *consist only of shapes from* S.

Interestingly, we found a correlation between streak length and query shape and size. For instance, out of the 526 streaks that contain a cyclic query, 472 (89.73%) only consist of a single query. This strongly contrasts the entire log, where only 189,319 streaks (37.10%) consist of a single query. Similarly, we have 1,378 streaks that contain a query of at least 16 triples, but 1,332 of these streaks (96.66%) only have a single query. This suggests that highly complex queries are less likely to

occur in longer streaks. We stress again that the data sets used for this study only consisted of DPpedia query logs for *three days*, which is a very small sample. We leave the evaluation on the total corpus for future work.

11 Conclusions and Discussion

We have conducted an extensive analytical study on a large corpus of real SPARQL query logs. Our corpus is inherently heterogeneous and consists of a majority of DBpedia query logs along with query logs on biological datasets (namely BioPortal and BioMed datasets), geological datasets (LGD), bibliographic data (SWDF), and query logs from a museum's SPARQL endpoint (British Museum). We have completed this corpus with the example queries from Wikidata (Feb. 2017), which are cherry picked from real SPARQL queries on this data source. Compared to the conference version of this paper, we have augmented the corpus with $169M$ queries from DBpedia, which let us almost double the size of the corpus and also corroborate or deflect some of the insights gained before on the old logs. Furthermore, novel non-trivial analyses have been run as also recapitulated in this concluding section.

A Note on Query Logs and Interpretation of Results. When one wants to draw conclusions from our analyses, one always needs to keep in mind what kind of data we analysed, in order to put the conclusions in the right perspective. In this paper, we mainly analysed *query logs from SPARQL endpoints*. We believe that this means that simple queries may be overrepresented. For instance, some users may decide to download a local copy of the database to their own server and process the complex queries locally, e.g., to avoid time-out issues with the public SPARQL endpoint.

Another point to keep in mind is that we believe that it is difficult to conclude from such a log analysis that certain types of queries are not interesting. Again, this is due to the *open-world* nature of the logs. There can be very interesting types of queries, that some users are highly interested in, but that are absent from the logs.

What one *can* discover in our analysis is classes of queries, or aspects (such as sequences of queries) that are interesting for future research. After all, the queries we studied here are indeed precisely the ones that have been submitted to SPARQL endpoints, which makes them interesting.

Considerations About the Datasets. The majority of the datasets exhibit similar characteristics, such as for instance the simplicity of queries amounting to 1 or 2 triples. The only exception occurs with British Museum and Wikidata datasets (Figure 1), where the former is a set of queries generated from fixed templates and the latter is a query *wiki* rather than a query log. Clearly, the DBpedia datasets are the most voluminous and recent in our corpus, thus making their results quite significant. For instance, despite the fact that single triple queries are numerous in these datasets, more complex queries (with 11 triples or more) have lots of occurrences (up to 21% of the total number of queries for DBpedia13). Strikingly, the largest queries of all belong to DBpedia (especially the last logs newly analyzed in this paper), which is one of the outcome of the new comparison between Valid and Unique queries, as carried out in this paper and could not be observed before in [11], which only focused on Unique queries.

We observed that most of the analyzed queries across all datasets are Select/Ask/Construct, which range between 94% and 100% for all datasets except DBpedia16, BioMed, and SWDF, which have 88% or less. Therefore, we focused on such queries in the remainder of the paper since these queries turn out to be the queries that users most often formulate in SPARQL query endpoints. We have further examined the occurrences of operator distributions and the number of projections and subqueries. This analysis lets us address a specific fragment, namely the And/Opt/Filter patterns (AOF patterns). For such patterns, we derived the graph- and hypergraph structures and analyzed the impact of the structure on query evaluation.

Benefits of Shape Analysis. We synthetically reproduced the observed real chain and cycle query logs with a synthetic generator by building diverse workloads of Ask queries and measured their average runtime in two systems, Blazegraph, used by the Wikimedia foundation, and PostgreSQL. In both systems, the difference between average performances of such different query shapes are perceivable. We decided to dig deeper in the shape analysis in order to classify these queries under general query shapes as canonical graphs and characterize their tree-likeness as hypergraphs. We believe that this shape analysis can serve the need of fostering the discussion on the design of new query languages for graph data [10, 44], as pursued for instance by the LDBC Graph Query Language Task Force [43]. It can also inspire the conception of novel query optimization techniques suited for these query shapes, along with tuning and benchmarking methods. For instance, we are not aware of existing benchmarks targeting flowers and flower sets. The analysis on property paths showed that these are not yet widely used in the entire corpus, even though they are numerous in the Wikidata corpus.

A recent discussion (July 6th, 2017) in a Neo4J working group [48] concerned the support of full-fledged regular path queries in OpenCypher. This discussion, and other discussions on standard graph query languages [43,10,44] could benefit from our analysis, devoted to find which property paths are actually used most often when ordinary users have the power of regular expressions. On both shape analysis and property path analysis, the addition of the new DBpedia17 logs provided us with both (1) confirmation of the trends observed before on a restricted corpus [11]; (2) new insights due to the injection of new logs. Concerning the shape analysis, we introduced a new class (no edge) leading to classify queries consisting of isolated nodes, such class being inflated when constants are disregarded in the analysis. Furthermore, the shapes of Valid queries studied for the first time in this paper are comparably more complex than the shapes of Unique queries. Precisely, we have observed that Valid queries exhibit on average longer paths and higher degree nodes compared to their Unique counterparts. For the property paths, we could confirm the presence of classes observed before with most occurrences but also introduce entirely new classes due to the presence of more diversified DBpedia query logs.

Benefits of Streak Analysis. Finally, we performed a study on the way users specify their queries in SPARQL query logs, by identifying streaks of similar queries. This analysis is for instance crucial to understand query specification from real users and thus usability of databases, which is a hot research topic in our community [25,38].

Extensibility. Our analysis has been carried out with scripts in different languages, amounting to a total of roughly 9,000 source lines of code (SLOC). We plan to make these scripts open-source and extensible to the new query logs that will be produced by users on SPARQL endpoints in the near future.

Future Work. A preliminary investigation on our data set showed that a shape analysis that incorporates property paths (and therefore considering extensions of CRPQs instead of CQs) may reveal interesting results. For instance, we found a 7-clique query (6-clique without constants) similar to the one in Figure 9. We also found this particular query interesting because we believe that its semantics is probably different from what the user intended. We believe that the user wanted to search for (possibly all permutations of) six different spouses of Henry VIII. However, "!dbpedia-owl:sameAs" tests if there exists an edge between two nodes that is not classified as dbpedia-owl:sameAs. We embarked on a study for Wikidata query logs in [12].

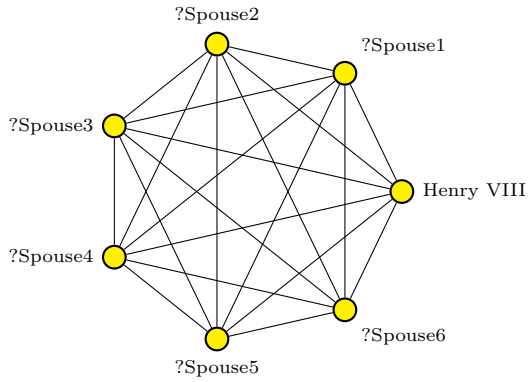


Fig. 9 The Henry VIII query, a 7-clique containing one constant and six variables. All edges between Henry VIII and the variables are labeled "dbpedia-owl:spouse" and all edges between variables are labeled with the property path "!dbpedia-owl:sameAs".

Acknowledgments

We would like to acknowledge USEWOD and Patrick van Kleef together with the team of OpenLink Software for hosting the official DBPedia endpoint and granting us the access to the large DBpedia query logs analysed in this paper. We thank Stijn Vansummeren for his suggestion to investigate free-connex acyclicity of queries.

References

1. C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. EmptyHeaded: A relational engine for graph processing. In *International Conference on Management of Data (SIGMOD)*, pages 431–446, 2016.
2. C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Old techniques for new join algorithms: A case study in RDF processing. In *International Conference on Data Engineering (ICDE) Workshops*, pages 97–102, 2016.
3. M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. *CoRR*, abs/1103.5043, 2011.
4. G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gmark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.*, 29(4):856–869, 2017.
5. G. Bagan, A. Bonifati, and B. Groz. A trichotomy for regular simple path queries on graphs. In *Principles of Database Systems (PODS)*, pages 261–272, 2013.
6. G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Computer Science Logic (CSL)*, pages 208–222, 2007.
7. M. Bannach, S. Berndt, and T. Ehlers. Jdrasil: A modular library for computing tree decompositions. In *16th International Symposium on Experimental Algorithms (SEA)*, pages 28:1–28:21, 2017.

8. P. Barceló, R. Pichler, and S. Skritek. Efficient evaluation and approximation of well-designed pattern trees. In *Principles of Database Systems (PODS)*, pages 131–144, 2015.
9. A. Bielefeldt, J. Gonsior, and M. Krötzsch. Practical linked data access via SPARQL: the case of wikidata. In *Workshop on Linked Data (LDOW)*, 2018.
10. A. Bonifati, G. H. L. Fletcher, H. Voigts, and N. Yakovets. *Querying Graphs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.
11. A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *PVLDB*, 11(2):149–161, 2017.
12. A. Bonifati, W. Martens, and T. Timm. Navigating the maze of Wikidata query logs. In *The World Wide Web Conference (WWW)*, pages 127–138, 2019.
13. A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Symposium on the Theory of Computing (STOC)*, pages 77–90, 1977.
14. C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *International Conference on Database Theory (ICDT)*, pages 56–70, 1997.
15. W. Czerwiński, W. Martens, M. Niewerth, and P. Parys. Minimization of tree patterns. *J. ACM*, 65(4):26:1–26:46, 2018.
16. W. Czerwiński, W. Martens, P. Parys, and M. Przybylko. The (almost) complete guide to tree pattern containment. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 117–130, 2015.
17. `detkdecomp`. `wwwinfo.deis.unical.it/~frank/Hypertrees`. Visited on August 10th, 2016.
18. G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree decompositions: Questions and answers. In *Principles of Database Systems (PODS)*, pages 57–74, 2016.
19. G. Gottlob, G. Greco, and F. Scarcello. Treewidth and hypertree width. In *Tractability: Practical Approaches to Hard Problems*, pages 3–38. Cambridge University Press, 2014.
20. G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.
21. X. Han, Z. Feng, X. Zhang, X. Wang, G. Rao, and S. Jiang. On the statistical analysis of practical SPARQL queries. In *WebDB*, page 2, 2016.
22. S. Harris and A. Seaborne. SPARQL 1.1 query language. Technical report, World Wide Web Consortium (W3C), March 2013. `https://www.w3.org/TR/2013/REC-sparql11-query-20130321`.
23. J. Huelss and H. Paulheim. What SPARQL query logs tell and do not tell about semantic relatedness in LOD — or: The unsuccessful attempt to improve the browsing experience of DBpedia by exploiting query logs. In *ESWC Satellite Events*, pages 297–308, 2015.
24. M. Idris, M. Ugarte, and S. Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *International Conference on Management of Data (SIGMOD)*, pages 1259–1274, 2017.
25. H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *International Conference on Management of Data (SIGMOD)*, pages 13–24, 2007.
26. O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. In *International Conference on Extending Database Technology (EDBT)*, pages 282–293, 2017.
27. M. Kaminski and E. V. Kostylev. Beyond well-designed SPARQL. In *International Conference on Database Theory (ICDT)*, pages 5:1–5:18, 2016.
28. B. Kimelfeld and Y. Sagiv. Revisiting redundancy and minimization in an XPath fragment. In *International Conference on Extending Database Technology (EDBT)*, pages 61–72, 2008.
29. M. Kröll, R. Pichler, and S. Skritek. On the complexity of enumerating the answers to well-designed pattern trees. In *International Conference on Database Theory (ICDT)*, pages 22:1–22:18, 2016.
30. A. Letelier, J. Pérez, R. Pichler, and S. Skritek. Static analysis and optimization of semantic web queries. *ACM Trans. Database Syst.*, 38(4):25:1–25:45, 2013.
31. L. Libkin, W. Martens, and D. Vrgoc. Querying graphs with data. *J. ACM*, 63(2):14:1–14:53, 2016.
32. 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 *International Semantic Web Conference (ISWC)*, pages 376–394, 2018.
33. W. Martens and J. Niehren. On the minimization of XML schemas and tree automata for unranked trees. *J. Comput. Syst. Sci.*, 73(4):550–583, 2007.
34. W. Martens and T. Trautner. Enumeration problems for regular path queries. *CoRR*, abs/1710.02317, 2017.
35. G. Miklau and D. Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 51(1):2–45, 2004.
36. K. Möller, M. Hausenblas, R. Cyganiak, S. Handschuh, and G. Grimnes. Learning from linked open data usage: Patterns & metrics. In *Web Science Conference (WSC)*, 2010.
37. M. Morsey, J. Lehmann, S. Auer, and A. N. Ngomo. DBpedia SPARQL benchmark — performance assessment with real queries on real data. In *International Semantic Web Conference (ISWC)*, pages 454–469, 2011.
38. A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *PVLDB*, 4(12):1466–1469, 2011.
39. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
40. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.
41. F. Picalausa and S. Vansummeren. What are real SPARQL queries like? In *International Workshop on Semantic Web Information Management (SWIM)*, pages 1–7, 2011.
42. M. Saleem, I. Ali, A. Hogan, Q. Mehmood, and A.-C. Ngonga Ngomo. LSQ: The linked SPARQL queries dataset. In *International Semantic Web Conference (ISWC)*, pages 261–269, 2015.
43. `http://ldbcouncil.org`.
44. `https://databasetheory.org/node/47`.
45. `http://wikidata.org`.
46. `http://wiki.dbpedia.org/datasets`.
47. `http://www.blazegraph.com`.
48. `http://www.opencypher.org/ocig2`.
49. `http://www.postgresql.org`.
50. M. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently joining group patterns in SPARQL queries. In *Extended Semantic Web Conference (ESWC)*, pages 228–242, 2010.
51. D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.