An Analytical Study of Large SPARQL Query Logs

Angela Bonifati[*]
Lyon 1 University
Lyon, France

Wim Martens
University of Bayreuth
Bayreuth, Germany

Thomas Timm
University of Bayreuth
Bayreuth, Germany

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 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 pseudographs, and characterize their (hyper-)tree 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.

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 [32, 33, 39]), 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 [24, 3, 28, 29, 15], 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., [9, 12, 14]) and is now being used for state-of-the-art structural decomposition methods for query optimization [1, 2, 18]. 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 [8, 7, 21]: 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 theory for over two decades [12] 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, 18]. 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 [12]. 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 [7]. For this rea-

---

[1]The exception is [15], where logs from the Linked SPARQL Queries Dataset (LSQ) were studied, combining data from four sources (from 2010 and 2014) that we also consider.

son, 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 small and simple and we believe this to be typical for SPARQL endpoint logs. For instance, the majority (>55%) of the queries in our logs only use one triple. 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 [5]. 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, 18].

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.

Related Work. Whereas several previous studies have focused on the analysis of real SPARQL queries, they have mainly looked at statistical features of the queries, such as occurrences of triple patterns, types of queries, query fragments and well-designed patterns [24, 3, 29, 15]. The only early study that investigated the relationship between structural features of practical queries and query evaluation complexity has been presented in [28]. 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. USE-WOD and DBpedia datasets have also been considered in [4]. It takes into account the log files from DBpedia and

Source	Total #Q	Valid #Q	Unique #Q
DBpedia9/12	28,534,301	27,097,467	13,437,966
DBpedia13	5,243,853	4,819,837	2,628,005
DBpedia14	37,219,788	33,996,480	17,217,448
DBpedia15	43,478,986	42,709,778	13,253,845
DBpedia16	15,098,176	14,687,869	4,369,781
LGD13	1,841,880	1,513,868	357,842
LGD14	1,999,961	1,929,130	628,640
BioP13	4,627,271	4,624,430	687,773
BioP14	26,438,933	26,404,710	2,191,152
BioMed13	883,374	882,809	27,030
SWDF13	13,762,797	13,618,017	1,229,759
BritM14	1,523,827	1,513,534	135,112
WikiData17	309	308	308
Total	180,653,910	173,798,237	56,164,661

Table 1: Sizes of query logs in our corpus.

SWDF reaching a total size of 3M. They mainly investigate the number of triples and joins in the queries. Based on the observation of [26] that typically SPARQL graph patterns are typically chains 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.

2. DATA SETS

Our data set has a total of $180,653,910$ queries, which were obtained as follows. We obtained the 2013–2016 USE-WOD query logs, DBPedia query logs for 2013, 2014, 2015 and 2016 directly from Openlink[3], the 2014 British Museum query logs from LSQ[4], and we crawled the user-submitted example queries from WikiData[5] 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 USE-WOD'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.[6] 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

---

[2] We do consider extensions with Filter and Opt, but only those for which we know that tree-likeness of their graph ensures the existence of efficient evaluation algorithms.

[3] http://www.openlinksw.com

[4] http://aksw.github.io/LSQ/

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

[6] We discovered that we received three log files from USE-WOD 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.

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 3.0.1 (*Valid*). From the latter set, we removed duplicate queries, resulting in the unique queries that we could parse (*Unique*) and on which we focus in the remainder of the paper [7]. 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), 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] In the total unique data set, 2,496,806 queries (4.47%) do not have a body. All these queries are Describe queries and almost exclusively occur in DBpedia14– DBpedia16.

3. PRELIMINARIES

We recall some basic definitions on RDF and SPARQL [27, 28]. We closely follow the exposition of [28].

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 distinction between IRIs, blank nodes, and literals is not important.

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

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

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 if 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 in $\mathcal{I}$ 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. With respect to [28], 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:

$$P ::= \quad t \mid pp \mid Q \mid P_1\ \mathsf{And}\ P_2 \mid P\ \mathsf{Filter}\ R$$
$$\mid P_1\ \mathsf{Union}\ P_2 \mid P_1\ \mathsf{Opt}\ P_2 \mid \mathsf{Graph}\ iv\ P$$

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. We refer to the SPARQL 1.1 recommendation [16] and the literature [27] 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 [32] ("Locations of archaeological sites", from [32]).

```
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).

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 3.1. *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 first block in Table 2 describes the type of queries. In total, 87.97% of the queries are Select-queries, 4.97% are Ask-queries, 4.59% Describe queries, and 2.47% Construct queries. There are, however, tremendous differences

---

[7]We report in a related appendix [37] the results for the *Valid* corpus, containing duplicates.

[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.

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

Element	Absolute	Relative
Select	49,409,913	87.97%
Ask	2,789,420	4.97%
Describe	2,578,311	4.49%
Construct	1,386,908	2.47%
Distinct	12,198,198	21.72%
Limit	9,545,249	17.00%
Offset	3,455,500	6.15%
Order By	1,159,231	2.06%
Filter	22,547,561	40.15%
And	15,863,942	28.25%
Union	10,465,706	18.63%
Opt	9,106,419	16.21%
Graph	1,519,899	2.71%
Not Exists	926,849	1.65%
Minus	766,380	1.36%
Exists	5,499	0.01%
Count	320,035	0.57%
Max	3,660	0.01%
Min	3,632	0.01%
Avg	263	< 0.01%
Sum	68	< 0.01%
Group By	168,444	0.30%
Having	12,276	0.02%

Table 2: Keyword count in queries

between the data sets. `BioMed13` has less than 13% Select-queries and almost 85% Describe-queries, whereas `LGD13` has 28% Select-queries and 71% Construct-queries. Even within the same kind of data, we see significant differences. `DBpedia16` has 62% Select-queries (and 34% Describe-queries), whereas `DBpedia15` has 81.5% Select-queries and 11.5% Ask-queries. The other DBpedia data sets have over 87.5% Select 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 (97%) of `BritM14` queries use Distinct. This is similar, but to a lesser extent in `BioP13` (82%) and `BioP14` (69%). In DBPedia we again see significant differences. From '12 to '16, we have 18%, 8%, 11%, 38%, and 8% of queries with Distinct respectively.

Limit is used most widely in `SWDF13` (47%) and `LGD14` (41%). The most prevalent data sets for queries with Offset are `LGD14` (38%), `LGD13` (13%), and `DBpedia13` (12%).

Order By is used by far the most in WikiData (42%), which may be due to the case that their queries are intended to showcase the system and should produce a nice output. Another reason may be that the other query logs 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 8).

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

---

[10]The remaining solution modifier, Reduced, was only found in 1.113 queries.

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

data, we see that 95% of the queries using Graph originate from `BioP13` and `BioP14`. In these logs, 80% and 40% of the queries use Graph, respectively. The use of Filter ranges from 61% (`LGD14`) to 3% or less (`BioMed13`, `BioP13`).

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) [16]. In all data sets, each of these operators was used in 3% or less of the queries, except for `LGD14` (31% with Count) and `WikiData17` (30% with Group By). We see a higher relative use of aggregation operators in `WikiData17` than in the other sets, which we believe may be due to the fact that our `WikiData17` set is not a query log. `WikiData17` is in fact a wiki page that contains cherry-picked and user-submitted queries, some of which are meant to highlight features of the Wikidata data set.

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 and Ask 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 under scrutiny later in the paper. We focus solely on Select and Ask queries because these are the type of SPARQL statements that truly query the data, as opposed to Describe statements (which are exploratory) and Construct statements (which construct data).[12]

The plot in the upper part of Figure 1 illustrates the results in terms of the percentages of Select and Ask queries (per dataset) containing respectively from 0 triples to a number of triples greater than 11. 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`. Both datasets have in fact unique characteristics, `BritM14` being a collection of queries with fixed templates and `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`–`DBpedia16`, along with `LGD14` and `BioMed13` are the datasets exhibiting the most complex queries with extremely high numbers of triples exceeding 11.

We should note that `BioMed13` has almost 85% Describe queries and 2.42% Construct queries. The numbers reported here only describe the remaining 12.87%. The table at the bottom of Figure 1 shows the relative amounts of Select- and Ask-queries per data set. It also shows the average number of triples measured across all queries within each dataset. We can notice a relative increase of this average for DPpedia from year 2014 up to year 2016 and BioPortal and LGD in between years 2013 and 2014. As expected, BioMed, BritishM and Wikidata have also relatively high average number of triples, compared to the other datasets for the reasons previously exposed.

---

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

Figure 1: Percentages of queries exhibiting different number of triples (in colors) for each dataset (top), the average numbers of triples of the queries for each dataset ($Avg_{\#T}$, bottom), and the percentage of Select/Ask-queries (S/A, bottom).
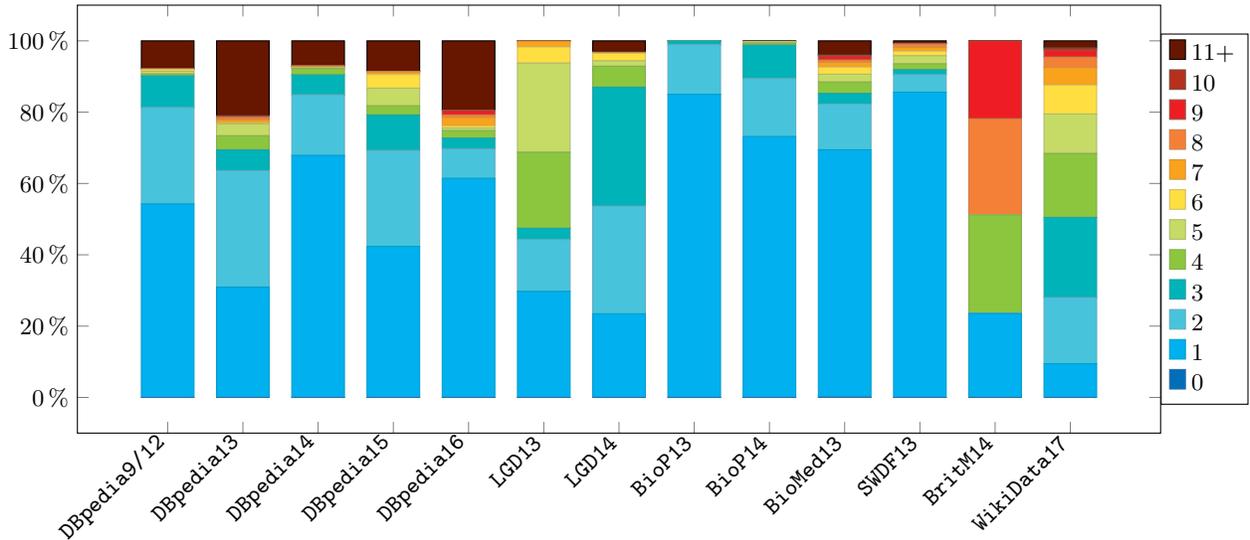
Datasets	DBpedia9/12	DBpedia13	DBpedia14	DBpedia15	DBpedia16	LGD13	LGD14	BioP13	BioP14	BioMed13	SWDF13	BritM14	WikiData17
S/A	99.15%	91.88%	95.38%	93.05%	63.99%	29.01%	97.47%	100%	99.69%	12.87%	96.14%	98.64%	99.68%
$Avg_{\#T}$	2.38	3.98	2.09	2.94	3.78	3.19	2.65	1.16	1.42	2.44	1.51	5.47	3.94

Operator Set	Absolute	Relative
none	17,482,313	33.49%
F	9,936,557	19.04%
A	3,911,748	7.49%
A, F	3,261,138	6.25%
CPF subtotal	31,330,554	66.27%
O	542,900	1.04%
O, F	1,791,512	3.43%
A, O	1,728,907	3.31%
A, O, F	406,131	0.78%
CPF+O	+4,469,450	+8.56%
G	1,380,764	2.65%
CPF+G	+1,432,090	+2.74%
U	3,895,524	7.46%
U, F	198,693	0.38%
A, U	817,958	1.57%
A, U, F	812,381	1.56%
CPF+U	+5,724,556	+10.97%
A, O, U, F	4,084,154	7.82%

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

Overall, we see that 56.45% of the Select and Ask-queries in our corpus use at most one triple, 90.76% uses at most six triples, and 99.32% at most twelve triples. The largest queries we found came from `DBpedia15` (209 and 211 triples) and `BioMed13` (221 and 229 triples).

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- and Ask queries. We then investigated how these operators occur together. In particular, we investigated for which queries

the body *only* uses constructs with these operators.[13] [14]

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 $\mathcal{O} = \{\text{Filter}, \text{And}, \text{Opt}, \text{Graph}, \text{Union}\}$ and, on its right, the amount of queries in our logs for which the body uses exactly the operators in S (and none from $\mathcal{O} \backslash S$). 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 [26, 38]

DEFINITION 4.1. A *conjunctive pattern with filters (CPF)* is a graph pattern that only uses triples and the operators And and Filter.

Our logs contain 66.27% CPF patterns. Adding Opt to the CPF fragment would increase its relative size with 8.56%, resulting in 74.83% of our queries. (Similarly for Graph and Union.)

Table 3 classifies 96.37% of the Select- and Ask queries in our corpus. The remaining queries either use other combinations from $\mathcal{O}$ (0.30%), use other features than those in $\mathcal{O}$ in their body (3.33%) like Bind, Minus, subqueries or property paths.

There is a close relationship between CPF patterns and *conjunctive queries* that, in some cases, can be extended to also include queries with Opt and Graph. We discuss this in more detail in Section 5.

---

[13]There is one exception: For Wikidata, we removed SERVICE subqueries before the analysis (which appears in 222 of its queries and is used to change the language of the output).

[14]This study closely follows a similar one [28] that was done on a log from DBpedia 2010. Our numbers should be compared to the numbers of ULog (the duplicate-free log) in [28].

4.4 Subqueries and Projection

Only 304,234 (0.54%) 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 [7, 22, 19, 27, 28] define evaluation as the following question: *Given an RDF graph G, a graph 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 [8, 7, 21], but its complexity drops to PTIME if projection is absent [27, 7, 22].[15] Therefore, the use of projection has a huge influence of the complexity of query evaluation.

Surprisingly, we discovered that at least 14.98% of the queries use projection, which is about three times more than what Picalausa and Vansummeren discovered in DBpedia logs from 2010 [28]. The 14.98% consists of 13.12% Select queries plus 1.86% Ask queries. Notice that the total number of Ask queries (4.97%) 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 [16], we classified these queries as not using projection.

Due to the use of the Bind operator, there was a number of queries (1.3%) where we could not determine if they use projection or not. Therefore the number of queries with projection lies between 14.98% and 16.28%.

5. STRUCTURAL ANALYSIS

SPARQL patterns of Select or Ask 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., [27, 19, 7, 20, 22]). We refer to such patterns as *And/Opt/Filter patterns* or, for succinctness, *AOF patterns.* Our corpus has 39.061.206 AOF patterns (74.83% of the Select- and Ask queries).

In Section 6 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 5.1) and we therefore need to turn to their hypergraph structure. Since the graph structure may be easier to understand, 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 [8], but becomes PTIME if their *hypertree width* is bounded by a constant [14]. Here, the hypertree width measures how close the query is to a tree (the lower the width,

$$x1 \xrightarrow{:a} x2 \xrightarrow{:b} x3 \xrightarrow{:c} x4$$

$$x1 \xrightarrow{x2} x3 \xrightarrow{:a} x4 \xrightarrow{x2} x5$$
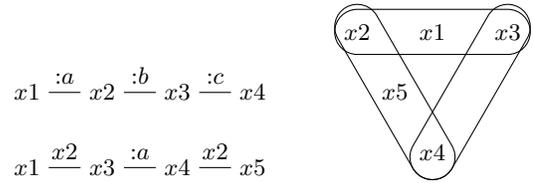
Figure 2: Canonical graphs and hypergraph for queries in Example 5.1.

the closer the query is to a tree). Several state-of-the-art join evaluation algorithms (e.g., [1, 18]) effectively use the hypergraph structure of queries to improve their performance, even in the context of RDF processing [2]. We establish in Section 5.1 that there are significant performance differences in today's query engines, even when the hypertreewidth of queries just increases from one to two.

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 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.[16] We define the *canonical graph* of graph pattern P to be the following graph: $E = \{\{x, y\}) \mid \ell$ is a literal and (x, ℓ, y) is a triple pattern in $P\}$ and $V = \{x \mid (x, \ell, y) \in E$ or $(y, \ell, x) \in E\}$.

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

EXAMPLE 5.1. *Consider the following (synthetic) queries:*

```
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, just as for the first query, a sequence of three edges, which completely ignores the join condition on ?x2. For this query, the canonical hypergraph in Figure 2 (right) correctly captures the cyclicity of the query.

5.1 Comparative Evaluation of Chain and Cycle Queries

---

[15]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.

[16]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.

We conducted a set of experiments aiming at comparing the execution times of conjunctive queries whose their corresponding 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 5.1 is a chain query of length three.) A *cycle query (of length k)* is a CQ for which the canonical graph is isomorphic to $\{x_0, x_1\}, \ldots, \{x_{k-1}, x_0\}$. 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., [18]. In order to generate query workloads containing the aforementioned types of queries, we have used gMark [5], a publicly available[17] schema-driven generator for graph instances and graph queries. We tuned gMark to generate diverse query workloads, each containing 100 chain and cycle queries, respectively.[18] 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 [36], an open-source relational DBMS, and BlazeGraph [34], an high-performance SPARQL query engine powering the Wikimedia's official query service [39] and thus used for Wikidata real-world queries. We have run these experiments on 2-CPUs Intel Xeon E5-2630v2 2.6 GHz server[19] 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 [5] 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 (and, vice versa for full workloads of Select queries, whose results are comparable and omitted for space reasons). 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 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

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

Figure 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).

in the shape analysis of our query logs, which we report in Section 6.

5.2 Classes of Queries for (Hyper)graphs

We now discuss the classes of queries for which we will investigate 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, which is why we only consider AOF patterns here. The simplest such queries are the CQs, which motivated the classical literature on query evaluation and hypertree structure [8, 14]. We discovered that 54.58% of the AOF patterns are CQs.

Next, we extend CQs with Filter and Opt 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 [7, 27].

We first define a fragment of CPF patterns that can be readily translated to CQs and can be evaluated similarly. We say that a filter constraint R is *simple* if vars(R) contains at most one variable or is of the form $?x = ?y$.[20] (An almost identical class of queries was also considered in [28].)

DEFINITION 5.2. A *conjunctive query with filters (CQ_F query)* is a CPF pattern that only uses simple filters.

In our corpus, 84.08% of the AOF patterns are in CQ_F.

We now additionally consider Opt. Pérez et al. [27] showed that unrestricted use of Opt in graph patterns makes query

---

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

[18] 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.

[19] Every CPU has 6 physical cores and (with hyperthreading) 12 logical cores.

---

[20] If we encounter a filter constraint of the form $?x = ?y$, we collapse the nodes $?x$ and $?y$ in the graph and hypergraph of the query.

T_1: $(\mathbf{?A}, \text{name}, ?N)$ T_2: $(\mathbf{?A}, \text{name}, ?N)$

$(\mathbf{?A}, \text{email}, ?E)$ $(\mathbf{?A}, \text{webPage}, ?W)$ $(\mathbf{?A}, \text{email}, ?E)$

$(\mathbf{?A}, \text{webPage}, ?W)$
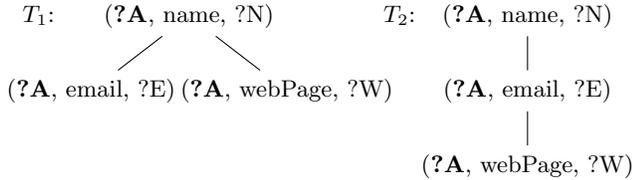
Figure 4: Pattern trees that correspond to the queries in Example 5.4

evaluation PSPACE-complete, which is significantly more complex than the NP-completeness of CQ_F queries. They discovered that patterns that satisfy an extra condition called *well-designedness* [27], can be evaluated much more efficiently. Letelier et al. show that, in the presence of projection, evaluation of well-designed patterns is Σ_2^P-complete [22].

DEFINITION 5.3. A graph 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 $\text{vars}(P_2) \setminus \text{vars}(P_1)$ occur in P only inside i.[21]

In our corpus, 98.53% 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 [7]. 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 [7] for details.

EXAMPLE 5.4. The following patterns come from [27, 22]:
$P_1 = (((?A, \text{name}, ?N) \text{ Opt } (?A, \text{email}, ?E))$

$\text{Opt } (?A, \text{webPage}, ?W))$

and $P_2 = ((?A, \text{name}, ?N)$

$\text{Opt } ((?A, \text{email}, ?E) \text{ Opt } (?A, \text{webPage}, ?W)))$

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 [23, 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* [27] and then turned into a pattern tree. The resulting pattern trees will then have a CQ in each of its nodes.

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 $(?A, \text{email}, ?E)$ 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.

---

[21]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.
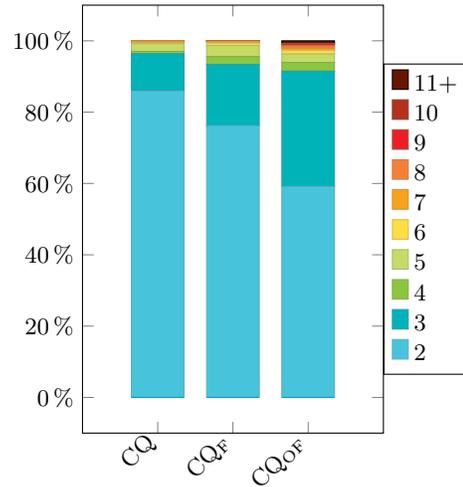
Figure 5: Size of CQ-like queries with at least two triples.

DEFINITION 5.5. A graph pattern P using only the operators And, Filter, and Opt is in CQ_{OF} if it has a well-designed pattern tree with interface width 1.

Perhaps surprisingly, out of all queries that are well-designed and have simple filters, we only found 310 queries that had an interface width more than one. In fact, 93.87% of the AOF patterns are CQ_{OF} queries.

6. SHAPE ANALYSIS

In this section we analyze the shapes of the canonical graphs and the tree- and hypertree width of CQs, CQ_F queries, and CQ_{OF} 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. The fractions of queries with one triple are 82%, 83.45%, and 75.52% for CQ, CQ_F, and CQ_{OF}, 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} queries with up to 229 triples.

6.1 Graph Structure

We analyse the graph structure of queries. Recall that we only consider graph shapes for queries that do not use variables in the predicate position of triples, for reasons explained in Section 5. We consider the remaining 6.96 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 5.1. A *chain set* is a graph in which every connected component is a chain. (So, each chain is also a chain set.)

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. A *forest* is a graph in which every connected component is a tree.

A *star* is a tree for which there exists exactly one node with more than two neighbors, that is, there is exactly 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, and CQ_{OF} queries. The last two fragments are interesting in that they bring under scrutiny more queries than the plain CQ set of query logs (by an increase of roughly 40% and 47%, 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 the three tables in Table 4. 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 very low occurrence with respect to the other tree-like shapes.

Since simple queries are overrepresented in query logs (already over 80% 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 6.1. 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*.

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 *flower sets*, to further increase the ratio of queries that could be classified from the original logs. The number of flowers and flower sets in the query logs overcome those of trees and forests by roughly 0.05%, respectively for CQ, CQ_F and CQ_{OF}, and for all the three fragments flowerSets queries could get significantly closer to 100% coverage than plain forests.

In the above analysis, we have analyzed the shapes of queries when the latter are represented as canonical graphs as defined in Section 5, i.e., the nodes can be either variables or constants. Constants are in fact necessary to fully characterize query shapes, even though they do not play a major role in query optimization, as variables do. For that reason, we have rerun the above analysis on queries excluding constants in order to identify the differences in the obtained
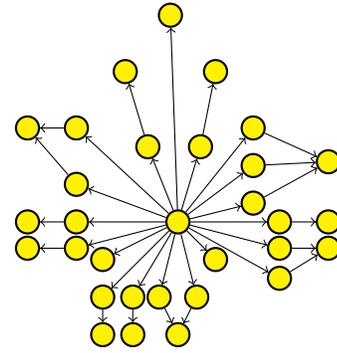
Figure 6: An example of flower query found in our DBPedia query logs (we added arrows to indicate the edge directions in the query; labels are omitted for confidentiality reasons).

shape classification. The most significant observation here was that 9.66 million single edge CQs (78.70% of the single edge CQs) uses constants.

For the queries with cycles, we also investigate what is the length of the *shortest cycle* in the query. We discovered, for 39,471 queries, the shortest cycle has length three. For 6,561 and 5,733 queries, the shortest cycles had length 4 and 5, respectively. For 26 queries, the length was larger. We found two queries for which the shortest cycle was 14, which is the largest value we found.

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 the CQs, CQ_F- and CQ_{OF} queries. We do not formally define tree- or hypertreewidth in this paper but instead refer to an excellent introduction [13]. In the terminology of Gottlob et al., we investigate 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 cycles, flowers, and flower sets have treewidth two. We investigated the remaining queries by hand and discovered that one query had treewidth three and all others had treewidth two, see Table 4. From the treewidth perspective, it is interesting to note that many queries of treewidth two are *flowers* or *flower sets* (Definition 6.1), which are a very restricted fragment.

Hypertree Width. We recall that we only considered canonical graphs for queries that do not use variables in the predicate position of triple patterns. In CQ_{OF}, 6,959,510 queries used this feature and therefore we must consider the hypergraph structure to correctly measure the cyclicity of these queries. We determined their (generalized) hypertree width with the tool `detkdecomp` from the Hypertree Decompositions home page [10].

Our results are as follows. All the remaining queries had hypertree width one, except for 86 queries with hypertree width two and eight queries with hypertree width three.

We also looked at the number of nodes in the hypertree

	CQ			CQ_F			CQ_OF	
Shape	*#Queries*	*Relative %*		*#Queries*	*Relative %*		*#Queries*	*Relative %*
single edge	12,273,871	77.98%		21,198,951	81.04%		21,479,706	72.30%
chain	15,561,944	98.87%		25,403,669	97.12%		26,887,865	90.50%
chain set	15,570,042	98.93%		25,418,689	97.17%		26,937,578	90.67%
star	147,457	0.94%		702,228	2.68%		2,654,497	8.94%
tree	15,723,163	99.90%		26,127,544	99.88%		29,599,539	99.63%
forest	15,731,535	99.95%		26,143,128	99.94%		29,651,600	99.81%
cycle	4,550	0.03%		4,705	0.02%		4,734	0.02%
flower	15,730,043	99.94%		26,135,676	99.92%		29,614,330	99.68%
flower set	15,738,439	100.00%		26,151,291	99.97%		29,666,423	99.86%
treewidth ≤ 2	15,739,056	100.00%		26,157,879	100.00%		29,708,967	100.00%
treewidth $= 3$	1	0.00%		1	0.00%		1	0.00%
total	15,739,057	100.00%		26,157,880	100.00%		29,708,968	100.00%

Table 4: Cumulative shape analysis of CQ, CQ_F, CQ_{FO} across all logs.
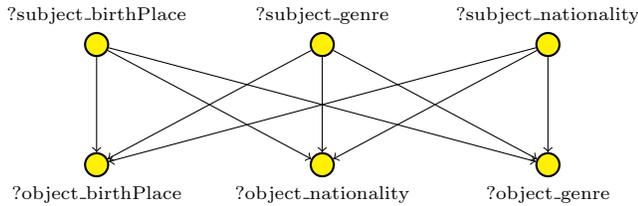
Figure 7: The DBPedia query exhibiting tree width equal to 3.

decompositions that the tool gave us, since this number can be a guide for how well *caching* can be exploited for query evaluation [18] (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 with more than 100 nodes in their hypertree decompositions, all of them occurring in `DBpedia15` and `DBpedia16`.) Finally, we observed that the queries with hypertree width two and three both had decompositions with up to ten nodes, respectively.

7. PROPERTY PATHS

We found 247,404 property paths in our corpus. 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.

A large fraction of these property paths are extremely simple. For instance, 63,039 property paths are !a ("follow an edge not labeled a") and 306 are ˆa ("follow an a-edge in reverse direction"). In the following, we focus on the remaining 184,059 property paths, which express queries on the graph that do more than simply follow an edge (such queries are sometimes called *navigational queries*).

Here, 66,262 (36%) use reverse navigation, i.e., the operator "ˆ", within more complex expressions. In Table 5, we present an overview of the property paths different from !a and ˆa. In our classification, we treat ˆa and !a the same as a literal. For instance, we classify a/b, $(ˆa)/b$, and $(!a)/b$ all as $a_1/\cdots/a_k$ with $k = 2$. When ! appears in front of a more complex expression (as in !$(a|b)$), we treat it separately. We

Expression Type	*Absolute*	*Relative*	*k*
$(a_1|\cdots|a_k)^*$	72,009	39.12%	2–4
a^*	48,636	26.42%	
$a_1/\cdots/a_k$	21,435	11.65%	2–6
a^*/b	19,126	10.39%	
$a_1|\cdots|a_k$	16,053	8.72%	2–6
a^+	3,805	2.07%	
$a_1?/\cdots/a_k?$	2,855	1.55%	1–5
$a(b_1|\cdots|b_k)$	37	0.02%	2
$a_1/a_2?/\cdots/a_k?$	31	0.02%	1–3
$(a/b^*)|c$	15	0.01%	
$a^*/b?$	13	0.01%	
$a/b/c^*$	11	0.01%	
!$(a|b)$	10	0.01%	
$(a_1|\cdots|a_k)^+$	10	0.01%	2
$(a_1|\cdots|a_k)(a_1|\cdots|a_k)$	5	< 0.01%	2–6
$a?|b$	2	< 0.01%	
$a^*|b$	2	< 0.01%	
$(a|b)?$	2	< 0.01%	
$a|b^+$	1	< 0.01%	
$a^+|b^+$	1	< 0.01%	
$(a/b)^*$	1	< 0.01%	

Table 5: Structure of navigational property paths in our corpus

only found 10 expressions that use ! and are different from the expression !a.

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 b/a^*. The variant listed in the table is the one that occurred most often in the data. That is, a^*/b occurred more often than b/a^*.

Bagan et al. [6] 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 one expression in our corpus which is not in C_{tract}, namely $(a/b)^*$.

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	26	8	357
71–80	15	4	129
81–90	5	1	54
91–100	4	0	27
>100	5	0	24

Table 6: Length of streaks in three single-day log files

8. 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 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 starting with q_1 and with q_2.

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%.[22] 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 keywords Select, Ask, Construct, or Describe. We took a window size of 30.

Since the discovery of streaks was extremely resource-consuming, we only analysed streaks in randomly selected log files from DBpedia14, DBpedia15, and DBpedia16. The sizes of these three 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.

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

---

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

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.

> **Wim:**
> Additional material:
>
> - The logs contain 34.137 streaks that have at least one valid query.
>
> - 1.614 have at least one erroneous query. (946 error → correct, 1.557 correct → error, 889 both).

9. 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) and geological datasets (LGD), query logs on bibliographic data (SWDF), and query logs from a museum SPARQL endpoints (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. 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). Moreover, we observed that most of the analyzed queries across all datasets are Select/Ask queries, which range between 91% and 99.88% for all datasets except DBpedia16 and LGD13, that have lower percentages. 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. We simulated 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, as pursued by the LDBC Graph Query Language Task Force [30, 11, 31]. 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 [35] concerned the support of full-fledged regular path queries in OpenCypher. This discussion, and other discussions on standard graph query languages [30, 11, 31] 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. Finally, we performed an 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 an hot research topic in our community [17, 25]. 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 publicly available in the next months.

Acknowledgments

10. 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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *International Conference on Database Theory (ICDT)*, pages 56–70, 1997.

[10] detkdecomp. wwwinfo.deis.unical.it/~frank/Hypertrees/. Visited on August 10th, 2016.

[11] G. H. L. Fletcher, H. Voigt, and N. Yakovets. Declarative graph querying in practice and theory. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, pages 598–601, 2017.

[12] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree decompositions: Questions and answers. In *Principles of Database Systems (PODS)*, pages 57–74, 2016.

[13] 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.

[14] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[15] 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.

[16] 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/.

[17] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 13–24, 2007.

[18] O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. In *International Conference on Extending Database Technology (EDBT)*, pages 282–293, 2017.

[19] M. Kaminski and E. V. Kostylev. Beyond well-designed SPARQL. In *International Conference on Database Theory (ICDT)*, pages 5:1–5:18, 2016.

[20] 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.

[21] 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, Dec. 2013.

[22] 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.

[23] 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.

[24] K. Möller, M. Hausenblas, R. Cyganiak, S. Handschuh, and G. Grimnes. Learning from linked open data usage: Patterns & metrics. In *Proceedings of the Web Science Conference*, 2010.

[25] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *PVLDB*, 4(12):1466–1469, 2011.

[26] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[27] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[28] F. Picalausa and S. Vansummeren. What are real sparql queries like? In *SWIM*, pages 7:1–7:6, New York, NY, USA, 2011. ACM.

[29] 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)*, 2015.

[30] `http://ldbcouncil.org/` .

[31] `https://databasetheory.org/node/47` .

[32] `http://wikidata.org` .

[33] `http://wiki.dbpedia.org/datasets` .

[34] `http://www.blazegraph.com` .

[35] `http://www.opencypher.org/ocig2` .

[36] `http://www.postgresql.org` .

[37] `http://www.theoinf.uni-bayreuth.de/download/sparql-rvzdb.pdf` .

[38] 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.

[39] D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.