What are real SPARQL queries like?

Francois Picalausa[*]
Université Libre de Bruxelles, Belgium
fpicalau@ulb.ac.be

Stijn Vansummeren[†]
Université Libre de Bruxelles, Belgium
stijn.vansummeren@ulb.ac.be

ABSTRACT

We present statistics on real world SPARQL queries that may be of interest for building SPARQL query processing engines and benchmarks. In particular, we analyze the syntactical structure of queries in a log of about 3 million queries, harvested from the DBPedia SPARQL endpoint. Although a sizable portion of the log is shown to consist of so-called *conjunctive* SPARQL queries, non-conjunctive queries that use SPARQL's UNION or OPTIONAL operators are more than substantial. It is known, however, that query evaluation quickly becomes hard for queries including the non-conjunctive operators UNION or OPTIONAL. We therefore drill deeper into the syntactical structure of the queries that are not conjunctive and show that in 50% of the cases, these queries satisfy certain structural restrictions that imply tractable evaluation in theory. We hope that the identification of these restrictions can aid in the future development of practical heuristics for processing non-conjunctive SPARQL queries.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Query languages; H.2.4 [**Database Management**]: Query processing

General Terms

Languages, Measurement, Theory

Keywords

SPARQL, conjunctive queries, log analysis, acyclicity

1. INTRODUCTION

The Resource Description Framework (RDF for short) provides a flexible method for representing information on the Semantic Web [13]. All data items in RDF are uniformly represented as triples of the form *(subject, predicate, object)*, sometimes also referred to as *(subject, property, value)* triples. Spurred by efforts like the Linking Open Data project [18], increasingly large volumes of data are being published in RDF.

RDF comes equipped with the SPARQL [12] language for querying data in RDF format. Using so-called *triple patterns* as building blocks, SPARQL queries search for specified patterns in the RDF data.

Many aspects of the challenges faced in large-scale RDF data management have already been studied in the database research community, including: native RDF storage layout and index structures [1, 5, 9, 15, 19]; SPARQL query processing and optimization [4,8,9]; as well as formal semantics and computational complexity of SPARQL [10, 14].

In this article, we present some statistics on real world SPARQL queries that may be of interest to researchers building SPARQL query processing engines and benchmarks. In particular, we analyze a log of SPARQL queries, harvested from the DBPedia SPARQL Endpoint from April to July 2010.[1] The log contains more than 3 million queries in total, of which over a million remain after duplicate elimination.

We analyze the syntactical structure of the queries in this log to get a feeling for the properties of the SPARQL queries posed in practice. In particular, we show that, although a sizable portion of the log consists of so-called *conjunctive* SPARQL queries, non-conjunctive queries that use SPARQL's UNION or OPTIONAL operators are more than substantial. Current proposals for native RDF query engines [8, 9, 17], however, focus their attention on providing innovative optimization algorithms for conjunctive SPARQL queries only, under the assumption that conjunctive queries are more commonly used than the others. By invalidating this assumption, our analysis hence motivates further research on practical heuristics for processing non-conjunctive SPARQL patterns.

Unfortunately, however, recent results by Perez et al. [10] and Schmidt et al. [14] show that query evaluation quickly becomes hard for patterns including the operators UNION or OPTIONAL. We therefore drill deeper into the syntactical structure of the non-conjunctive queries in the log and show that in 50% of the cases, these patterns satisfy certain structural restrictions that imply tractable evaluation in theory. We hope that the identification of these restrictions can aid in the future development of practical heuristics for process-

[*]Supported by a FNRS/FRIA scholarship.
[†]Supported by the OSCB project funded by the Brussels Capital Region (contract PFS-ICT "OSCB").

[1]DBPedia (`http://dbpedia.org`) is a version of the well-known WikiPedia Encyclopedia in RDF format.

ing non-conjunctive SPARQL patterns.

Finally, we also provide detailed statistics on other SPARQL features including query result forms, solution modifiers, etc.

Related Work. Möller et al [7] were the first to analyze a log of SPARQL queries, studying (1) the types of SPARQL queries posed (SELECT, ASK, CONSTRUCT, DESCRIBE) and (2) the forms of triple patterns posed in practice. In contrast, our work focuses on identifying reasonable structural restrictions that limit evaluation complexity. After the acceptance of this article, it was brought to our attention that, in parallel to our work, Arias et al. [3] have also observed that non-conjunctive operators appear non-negligibly often in practical queries. In addition, they provide detailed statistics on the types of joins appearing in practical queries, but do not consider structural restrictions.

Paper Organization. This paper is further organized as follows. We provide the necessary background on RDF and SPARQL in Section 2. We then begin our analysis by a study of the basic query features (query types, solution modifiers, etc.). Finally, we analyze the syntactic structure of the patterns that occur in SPARQL queries in Section 4.

2. PRELIMINARIES

In this section, we present the necessary background on RDF and the syntax and semantics of SPARQL, following the notation and exposition of Pérez et al [10].

RDF. Data in RDF is built from three disjoints sets $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$ of *IRIs*, *blank nodes*, and *literals*, respectively. For convenience we will use shortcuts like $\mathcal{IBL}$ and $\mathcal{IB}$ to denote the unions $\mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ and $\mathcal{I} \cup \mathcal{B}$, respectively.

All information in RDF is represented by triples of the form (s, p, o), where s is called the *subject*, p is called the *predicate*, and o is called the *object*. To be valid, it is required that $s \in \mathcal{IB}$; $p \in \mathcal{I}$; and $o \in \mathcal{IBL}$.

Each set of RDF triples can easily be represented graphically as an edge-labeled graph in which the subjects and objects form the nodes, and edges are labeled by the corresponding predicates. For this reason, sets of RDF triples are also called *RDF graphs*. An *RDF dataset* D is a pair $D = (G, \gamma)$ with G an RDF graph and γ a function that assigns an RDF graph $\gamma(i)$ to each i in a finite set $dom(\gamma) \subseteq \mathcal{I}$ of IRIs.

SPARQL. Abstractly speaking, a SPARQL query Q is a 4-tuple of the form

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

At the heart of Q lies the *graph pattern* P that searches for specific subgraphs in the input RDF dataset. Its result is a (multi-)set of *mappings*, each of which associates variables to elements of $\mathcal{IBL}$. The *dataset-clause* is optional and specifies the input RDF dataset to use during pattern matching. If it is absent, the query processor itself determines the dataset to use. The optional *solution-modifier* allows sorting of the mappings obtained from the pattern matching, as well as returning only a specific window of mappings (e.g., mappings 1 to 10). The result is a list L of mappings. The actual output of the SPARQL query is then determined by the *query-type*:

- SELECT queries return projections of mappings from L (in order);
- ASK queries return a boolean: true if the graph pattern P could be matched in the input RDF dataset, and false otherwise;
- CONSTRUCT queries construct a new set of RDF triples based on the mappings in L; and
- DESCRIBE queries return a set of RDF triples that describes the IRIs and blank nodes found in L. The exact contents of this description is implementation-dependent.

We will not further describe the syntax and semantics of *query-type*; *dataset-clause*; and *solution-modifier* here, but refer instead to the SPARQL recommendation [12] for those components.

For the analysis and results of Section 4, however, we do require a formal definition of SPARQL graph patterns. Following the example of Pérez et al [10], we do not use the concrete syntax of SPARQL for this purpose, but introduce an abstract syntax that is easier to use in the formal definitions that follow. All SPARQL queries in concrete syntax can be represented in this abstract syntax in a straightforward manner (see also Example 1 below).

Graph Patterns. Let $\mathcal{V} = \{?x, ?y, ?z, \dots\}$ be a set of variables, disjoint from the sets $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$. A *triple pattern* is an element of $(\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{L} \cup \mathcal{V})$. A *graph pattern* is an expression that can be generated by the following grammar:

$$\begin{array}{rcl} P & ::= & t \mid P_1 \text{ AND } P_2 \mid P_1 \text{ UNION } P_2 \mid P_1 \text{ OPT } P_2 \\ & \mid & P \text{ FILTER } R \mid \text{GRAPH } i\, P \mid \text{GRAPH } ?x\, P \end{array}$$

Here, t ranges over triple patterns, i ranges over IRIs in $\mathcal{I}$, and $?x$ ranges over variables in $\mathcal{V}$. R ranges over SPARQL filter constraints. We refer to the SPARQL recommendation for the syntax and semantics of such constraints [12]. Parentheses may be added to SPARQL patterns to avoid ambiguities. In what follows, we write $vars(P)$ to denote the set of variables occurring in P.

To define the semantics of SPARQL graph patterns we need to introduce some terminology. A *mapping* μ is a partial function $\mu \colon \mathcal{V} \to \mathcal{IBL}$ that assigns values in $\mathcal{IBL}$ to a finite set of variables. The domain of μ, denoted by $dom(\mu)$ is the subset of $\mathcal{V}$ where μ is defined. Two mappings μ_1 and μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, when for all common variables $?x \in dom(\mu_1) \cap dom(\mu_2)$ it is the case that $\mu_1(?x) = \mu_2(?x)$. Clearly, if μ_1 and μ_2 are compatible, then $\mu_1 \cup \mu_2$ is again a mapping. Abusing notation we denote by $\mu(t)$ the triple obtained by replacing the variables in triple pattern t according to μ.

Analogously to [10] we define join, union, difference, and left outer join on two sets of mappings Ω_1 and Ω_2 as follows:

$$\begin{aligned} \Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\} \\ \Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\} \\ \Omega_1 - \Omega_2 &= \{\mu_1 \in \Omega_1 \mid \mu_1 \not\sim \mu_2 \text{ for all } \mu_2 \in \Omega_2\} \\ \Omega_1 ⟕ \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 - \Omega_2) \end{aligned}$$

Semantically, each graph pattern P evaluates to a set of mappings $[\![P]\!]_D$ when evaluated on an RDF dataset $D = (G, \gamma)$. This semantics is inductively defined as follows. Let $\mu \models R$ denote that μ satisfies filter condition R, as defined in [12].

$$
\begin{aligned}
[\![t]\!]_D &= \{\mu \mid dom(\mu) = vars(t) \text{ and } \mu(t) \in G_D\} \\
[\![P_1 \text{ AND } P_2]\!]_D &= [\![P_1]\!]_D \bowtie [\![P_2]\!]_D \\
[\![P_1 \text{ UNION } P_2]\!]_D &= [\![P_1]\!]_D \cup [\![P_2]\!]_D \\
[\![P_1 \text{ OPT } P_2]\!]_D &= [\![P_1]\!]_D ⟕ [\![P_2]\!]_D \\
[\![P \text{ FILTER } R]\!]_D &= \{\mu \in [\![P]\!]_D \mid \mu \models R\} \\
[\![\text{GRAPH } i\, P]\!]_D &= \{\mu \in [\![P]\!]_{(\gamma(i),\gamma)} \mid i \in dom(\gamma)\} \\
[\![\text{GRAPH } ?x\, P]\!]_D &= \bigcup_{i \in dom(\gamma)} \{?x \mapsto i\} \bowtie [\![P]\!]_{(\gamma(i),\gamma)}
\end{aligned}
$$

EXAMPLE 1. The official SPARQL syntax considers operators `UNION`, `OPTIONAL`, `FILTER` and *concatenation* via a point symbol (.) to construct graph patterns, which corresponds to our UNION, OPT, FILTER, and AND operators, respectively. In addition, the syntax considers `{ }` to group patterns, and include implicit rules of precedence and association. In particular, FILTER expressions apply to the whole group in which they occur and may express theta-joins in optionals. As such, the pattern of the SPARQL query

```
SELECT ?x ?y
WHERE ?x foaf:familyName "Smith" .
      ?x foaf:age ?age .
      OPTIONAL { ?x foaf:homepage ?y
                 FILTER (?age >= 25) }
```

would be correspondingly represented in our syntax by

$$
\Big([(?x, \text{foaf:familyName}, \text{"Smith"}) \text{ AND } (?x, \text{foaf:age}, ?age)] \\
\text{OPT } (?x, \text{foaf:familyName}, \text{"Smith"})\Big) \\
\text{FILTER}(!\text{BOUND}(?y) \vee ?age \geq 25)
$$

The translation of the former into our syntax is done in our analysis software similarly to the translation of the concrete SPARQL syntax into the *SPARQL algebra* as described in the W3C Recommendation [12].

3. ANALYSIS OF QUERIES

In what follows, we analyze a log of SPARQL queries harvested from the DBPedia SPARQL Endpoint from April to July 2010. The log contains 3130177 queries in total (after removal of syntactically invalid queries) posed by both humans and software robots. After removal of duplicate queries, 1343922 queries remain.

We denote by *Log* the entire query log, and by *ULog* the duplicate-free query log. We will normally give statistics with respect to both logs as we feel that both are interesting: statistics with respect to *Log* give a better insight into the *query workload* that *Log* represents, while statistics with respect to log give better insight into how certain features are used irrespective of workload.

When reporting amounts, we use shortcuts like "200 / 150 have X" to denote that 200 queries in Log have property X while 150 queries in ULog have property X.

In this section we analyze the structure of the SPARQL *queries* in the log, and in particular of their *query-type*, *dataset-clause*, and *solution-modifier* components (see Section 2). The SPARQL graph patterns of the queries are analyzed in Section 4.

Query types. The four different types of SPARQL queries (SELECT, ASK, CONSTRUCT, DESCRIBE) are distributed as follows in the log:

Type	# in Log	# in ULog
SELECT	2937330 (93.83%)	1295306 (96.35%)
ASK	171169 (5.47%)	39563 (2.94%)
CONSTRUCT	18431 (0.59%)	8179 (0.61%)
DESCRIBE	3247 (0.10%)	874 (0.07%)

Perhaps unsurprisingly, most queries in the log are SELECT queries, although a sizable amount of boolean ASK queries also occur. The amount of CONSTRUCT and DESCRIBE queries is negligible. A possible explanation for the fact that so few DESCRIBE queries occur may be that the semantics of DESCRIBE depends on the SPARQL processor, which makes it difficult for a user to know when such a query is useful. A similar observation concerning the distribution of query types has been made by Möller et al. [7].

Duplicate elimination. Of the 2937330/ 1295306 SELECT queries, 1123060 (38.23%)/ 301715 (23.29%) queries use the DISTINCT keyword to indicate that duplicates must be removed from the query result, while only 3686 (0.13%)/ 1005 (0.08%) queries use the REDUCED keyword to indicate that it is permitted (but not required) to remove duplicates. Note that these keywords are not allowed in the other query types. The rationale behind REDUCED is that it may help query optimizers to find more efficient query plans. Its negligible use in practice indicates that query processors should not count on being able to exploit it, however.

Projection. Of the SELECT queries, 2698119 (91.86%)/ 1232225 (95.13%) output all variables mentioned in their graph pattern, i.e., they are of the form

$$\text{SELECT } \overline{x} \textit{ dataset-clause } P \textit{ solution-modifier}$$

with $\overline{x} = vars(P)$. This indicates that practical queries do not introduce "temporary" variables that are not output but only used to constrain the possible values of the real output variables. (For example, $?age$ in Example 1 is such a temporary variable that constrains only the legal bindings for $?x$ and $?y$.)

Dataset specification. Only 406418 (12.98%) / 309951 (23.06%) queries use a *dataset-clause* with a FROM (406377 / 309925) or FROM NAMED (57 / 37) specifier. The rest of the queries does not specify a *dataset-clause* and hence evaluates their graph pattern with respect to the processor-defined RDF dataset.

Sorting and windowing. SELECT, CONSTRUCT and ASK queries can use the ORDER BY, LIMIT, and OFFSET solution modifiers to sort the pattern matching results, or select only a specific window of these results. These modifiers are not used very often, as the following table shows:

Modifier	# in Log	# in ULog
ORDER BY	32470 (1.04%)	10238 (0.76%)
LIMIT	137223 (4.38%)	47248 (3.52%)
OFFSET	35780 (1.14%)	8711 (0.65%)

4. ANALYSIS OF GRAPH PATTERNS

4.1 Use of Operators

Let us next analyze the structure of the patterns in the log. In this respect, table 1 shows for each pattern operator $o \in \{\text{AND}, \text{FILTER}, \text{OPT}, \text{UNION}, \text{GRAPH}\}$ the number of patterns in which o occurs. As such, it gives an idea of how frequent an operator is used in practical patterns.

Operator o	# in Log		# in ULog	
AND	1294364	(41.34%)	665708	(49.53%)
FILTER	939740	(30.02%)	535450	(39.84%)
OPT	925460	(29.57%)	623370	(46.38%)
UNION	653303	(20.87%)	361251	(26.88%)
GRAPH	43	(0.00%)	27	(0.00%)

Table 1: Use of individual operators.

We see in particular that all operators except GRAPH occur non-negligibly often. Table 1 is complemented by Table 2 which lists, for each set of pattern operators O, the number of patterns in the log that use *exactly* all operators in O. (For space reasons, we omit those O for which this number is zero.) As such, Table 2 gives an idea of how frequently operators appear in combination.

Triple Patterns. The first row of Table 2 counts the number of patterns that do not use any operator (i.e., $O = \emptyset$). These patterns necessarily consist of a single triple pattern and hence express a simple selection on the input RDF dataset. Note that these patterns occur most frequent of all operator sets (45.30%/33.85%). The second row of Table 2 counts the number of patterns that use only the operator FILTER (7.70%/10.30%). These patterns are necessarily of the form t FILTER R and hence also express a simple selection on the input RDF dataset, where the variables in triple pattern t are constrained by R. The high frequency of operator sets $\emptyset$ and {FILTER} justifies the ongoing research in native RDF index structures that are specifically aimed at efficiently supporting selections.

Conjunctive versus non-conjunctive patterns. Current research in SPARQL pattern processing (e.g., [8,9,17]) focuses on optimizing the class of so-called *conjunctive* patterns (possibly with filters) under the assumption that these patterns are more commonly used than the others.

DEFINITION 2. *A* conjunctive pattern with filters *(CPF pattern for short) is a pattern that uses only the operators* AND *and* FILTER*.*

From Table 2 we see, however, that although a sizable portion of the log (65.68% / 51.25%) consists of CPF patterns, non-CPF patterns that use SPARQL's UNION and OPT operators are more than substantial (34.14% / 48.75%).

While UNION and OPT correspond to the relational database operators OUTER UNION and LEFT OUTER JOIN, respectively, it has been noted by Perez et al. [10] that optimization techniques developed for the latter in relational DBMSs cannot be applied to SPARQL patterns involving the former. Given the importance of UNION and OPT in practical patterns, we hence feel that more research towards practical heuristics for processing non-conjunctive patterns is required. The observations of the following section may aid in the development of these heuristics.

4.2 Structural properties and complexity

Clearly, we cannot hope to efficiently compute the set $[\![P]\!]_D$ of all answers to P unless the following decision version of the evaluation problem can be solved efficiently.

EVALUATION[O]

Input:	Pattern P that uses only operators in O; RDF dataset D; candidate mapping μ
Problem:	Decide whether $\mu \in [\![P]\!]_D$.

Recent results by Perez et al. [10] and Schmidt et al. [14] show, however, that the EVALUATION[O] quickly becomes hard when the set of operators O includes UNION or OPT. (Note that we here refer to the *combined* complexity of query evaluation [16].) The complexity of the problem as established in [10, 14] is shown, for each operator set O, in the last column of Table 2. In particular, EVALUATION becomes PSPACE-hard when OPT $\in O$, and NP-hard when $\{\text{AND}, \text{UNION}\} \subseteq O$.

PROVISO 1. All of the complexity upper bounds mentioned in this section hold under the assumption that checking whether a mapping μ satisfies a FILTER constraint R can be done in polynomial time (in the size of μ and R). This is certainly true for the SPARQL constraints that are composed entirely of the SPARQL built-in constraint operator functions. In general, however, FILTER constraints can also invoke arbitrary external functions, including those that do not have polynomial-time complexity. In the log, however, only 10849 / 2225 queries use a filter condition with external functions. After careful inspection, all of these external functions turned out to be polynomial-time computable.

It turns out that patterns often satisfy additional syntactical or structural restrictions that lower the complexity of EVALUATION. We introduce and discuss these restrictions next.

Well-designedness. Perez et al. [10] show that EVALUATION goes from PSPACE-hard to coNP-hard for patterns with OPT if they satisfy the following well-designedness condition.

DEFINITION 3. *A pattern P is* OPT*-well-designed (OWD for short) if for every subpattern $P' = (P_1$ OPT $P_2)$ of P and for every variable $?x$ that occurs in P the following condition holds: if $?x$ occurs both inside P_2 and outside P', then $?x$ also occurs in P_1.*[2] *A pattern P is in* union normal form *(UNF) if it is of the form*

$$P_1 \text{ UNION } P_2 \text{ UNION } \dots \text{ UNION } P_n$$

with $n \geq 1$ and every P_i UNION*-free.*

THEOREM 4 ([10]). EVALUATION *is* coNP*-complete for* GRAPH*-free patterns that are both in UNF and OWD.*

Out of the 925443 / 623358 GRAPH-free patterns that use OPT, 410586 (44.37%) / 297948 (47.80%) are both in UNF and OWD. For them, EVALUATION is hence less complex.

In fact, we found that a significant number of the patterns that use OPT and/or UNION satisfy an even stronger restriction, called *well-behavedness*, that ensures EVALUATION in PTIME. To the best of our knowledge, this restriction and its PTIME complexity is new. We introduce it next.

Well-behavedness. The first part of the well-behavedness restriction is due to Polleres [11].

[2]Strictly speaking, Perez et al. also require P to satisfy a certain *safety* condition, but the omission of this condition does not affect the results mentioned in this paper.

Operator set O	**# in Log**		**# in ULog**		Evaluation[O] **complexity**
none	1417583	(45.30%)	454891	(33.85%)	PTime
FILTER	241125	(7.70%)	138382	(10.30%)	PTime
AND	360027	(11.50%)	74908	(5.57%)	PTime
AND, FILTER	42734	(1.36%)	20643	(1.54%)	PTime
CPF subtotal	2061469	(65.85%)	688824	(51.25%)	
OPT	8761	(0.28%)	8044	(0.60%)	PSpace-complete
UNION	29998	(0.96%)	4495	(0.33%)	PTime
GRAPH	23	(0.00%)	12	(0.00%)	*not studied*
AND, OPT	315108	(10.07%)	240299	(17.88%)	PSpace-complete
AND, UNION	46503	(1.49%)	15001	(1.12%)	NP-complete
AND, GRAPH	3	(0.00%)	3	(0.00%)	*not studied*
FILTER, OPT	53585	(1.71%)	31066	(2.31%)	PSpace-complete
FILTER, UNION	35598	(1.14%)	7241	(0.54%)	PTime
OPT, UNION	938	(0.03%)	88	(0.01%)	PSpace-complete
AND, FILTER, OPT	37908	(1.21%)	14411	(1.07%)	PSpace-complete
AND, FILTER, UNION	31123	(0.99%)	4976	(0.37%)	NP-complete
AND, OPT, UNION	11491	(0.37%)	10729	(0.80%)	PSpace-complete
AND, OPT, GRAPH	2	(0.00%)	2	(0.00%)	PSpace-complete
FILTER, OPT, UNION	48202	(1.54%)	33995	(2.53%)	PSpace-complete
AND, FILTER, OPT, UNION	449450	(14.36%)	284726	(21.18%)	PSpace-complete
AND, FILTER, OPT, GRAPH	15	(0.00%)	10	(0.00%)	PSpace-hard
Non-CPF subtotal	1068708	(34.15%)	655098	(48.74%)	

Table 2: Distribution of sets of operators.

Definition 5. *A pattern P is* UNION-*well-designed (UWD for short) if for every subpattern $P' = (P_1$ UNION $P_2)$ of P and for every variable $?x \in vars(P')$ the following condition holds: if $?x$ occurs outside P', then $?x \in vars(P_1) \cap vars(P_2)$.*

For example, $[(?x, i, ?y)$ UNION $(i', ?y, ?z)]$ AND $(?y, ?v, l)$ is UWD, but $[(?x, i, ?y)$ UNION $(?v, ?y, ?z)]$ AND $(?y, ?v, l)$ is not. Out of the 653303 / 361251 patterns that use UNION, 652650 (99.90%) / 361141 (99.97%) are UWD (the patterns not using UNION are trivially UWD).

The second part of the restriction transfers the well-known notion of acyclicity for relational select-project-join queries [2] to SPARQL patterns.

Definition 6. *Let P be a CPF pattern. Let T be the set of all triple patterns occurring in P. The pattern P is said to be* acyclic *if it has a join forest. A join forest is a forest G (in the graph-theoretical sense) whose set of nodes is exactly T such that, for each pair of triple patterns t_1 and t_2 in T that have variables in common the following two conditions hold:*

1. *t_1 and t_2 belong to the same connected component of G; and*
2. *all variables common to t_1 and t_2 occur in every triple pattern on the (unique) path in G from t_1 to t_2.*

Efficient algorithms for checking acyclicity exist [2].

The final restriction concerns the filters constraints that can be used in acyclic patterns. Hereto, we introduce the following definition.

Definition 7. *A* FILTER *constraint R is* simple *if $vars(R)$ is either empty or consists of a single variable.*

Hence, constraints like LANGMATCHES$(?x, $"en"$)$ are simple, but constraints like $?x+?y \leq 50$ are not. Since a simple constraint R can constrain at most one variable, we can always push FILTER R to the triple pattern(s) in a given pattern P that mention the unique variable in $vars(R)$. This is not always possible for non-simple constraints as they can actually introduce join conditions. For example, in the pattern

$$[(?x, i, ?z) \text{ AND} (i', i'', ?y)] \text{ FILTER}(?x+?y < 10)$$

the filter cannot be pushed to either of the two triple patterns.

Definition 8. *A pattern P is* well-behaved *if (1) it is UWD; (2) it is OWD; and (3) for every pattern P_1* OPT *P_2 it holds that P_2 is an acyclic CPF pattern that mentions only simple filter constraints.*

Note in particular that well-behavedness prevents nesting of optionals: P_1 OPT $(P_2$ OPT $P_3)$ is not well-behaved. Also note that queries that use neither OPT nor UNION are trivially well-behaved.

Out of the 1068665/655086 patterns in the log that use OPT or UNION, 560694(56.07%)/328162(50.09%) are well-behaved. In total, 2622189(83.77%)/1017001(75.67%) of the patterns in the log are GRAPH-free and well-behaved. From the following theorem it hence follows that for the majority of queries in the log, Evaluation is in PTime.

Theorem 9. *Evaluation is in* PTime *for well-behaved patterns.*

Proof Sketch. Let P be a well-behaved graph pattern, let $D = (G, \gamma)$ be a dataset, and let μ be a candidate mapping. Polleres [11] shows that for patterns that are OWD and UWD, the notion of compatibility ($\mu_1 \sim \mu_2$) used when evaluating patterns can be strengthened. His result [11, Proposition 3] implies that $\mu \in [\![P]\!]_D$ can be inductively checked as follows. Let, for a set of variables X, $\mu[X]$ denote the restriction of μ to $dom(\mu) \cap X$. Let $X_1 = vars(P_1)$, and $X_2 = vars(P_2)$.

$$\mu \in [\![t]\!]_D \Leftrightarrow dom(\mu) = vars(t) \text{ and } \mu(t) \in G$$

$$\mu \in [\![P_1 \text{ AND } P_2]\!]_D \Leftrightarrow dom(\mu) \subseteq X_1 \cup X_2,$$
$$\mu[X_1] \in [\![P_1]\!]_D, \text{ and } \mu[X_2] \in [\![P_2]\!]_D$$

$$
\begin{aligned}
\mu \in [\![P_1 \text{ UNION } P_2]\!]_D &\Leftrightarrow dom(\mu) \subseteq X_1 \cup X_2 \text{ and either} \\
&\quad \mu[X_1] \in [\![P_1]\!]_D \text{ or } \mu[X_2] \in [\![P_2]\!]_D \\
\mu \in [\![P_1 \text{ FILTER } R]\!]_D &\Leftrightarrow \mu \in [\![P_1]\!]_D \text{ and } \mu \models R \\
\mu \in [\![\text{GRAPH } i\, P_1]\!]_D &\Leftrightarrow i \in dom(\gamma) \text{ and } \mu \in [\![P_1]\!]_D \\
\mu \in [\![\text{GRAPH } ?x, P_1]\!]_D &\Leftrightarrow \{?x\} \subseteq dom(\mu) \subseteq X_1 \cup \{?x\}, \\
&\quad \mu(?x) \in dom(\gamma), \text{ and} \\
&\quad \mu[X_1] \in [\![P_1]\!]_{(\gamma(\mu(?x)),\gamma)}
\end{aligned}
$$

Finally, $\mu \in [\![P_1 \text{ OPT } P_2]\!]_D \Leftrightarrow dom(\mu) \subseteq X_1 \cup X_2$ and $\mu[X_1] \in [\![P_1]\!]_D$, and either:

1. $\mu[X_2] \in [\![P_2]\!]_D$, or
2. $dom(\mu) \cap (X_2 - X_1) = \emptyset$ and $\mu[X_1 \cap X_2] \notin \pi_{X_1 \cap X_2}[\![P_2]\!]_D$

Here, $\pi_{X_1 \cap X_2}[\![P_2]\!]_D$ denotes the set $\{\mu'[X_1 \cap X_2] \mid \mu' \in [\![P_2]\!]_D\}$.

Clearly, this inductive reasoning runs in time polynomial in the size of P, D, and μ, provided that

$$\mu[X_1 \cap X_2] \notin \pi_{X_1 \cap X_2}[\![P_2]\!]_D \qquad (1)$$

can be checked in polynomial time. Fortunately, since P_2 is acyclic and uses only simple constraints, classical results from relational database theory imply that this is indeed the case [2, 6]. In particular, we reason as follows.

First observe that since P_2 uses only simple constraints, all filter constraints of the form FILTER R can be pushed down to the triple patterns in P_2 that mention the unique variable in $vars(R)$. In other words, we can rewrite P_2 into a pattern of the form $P_2' = Q_1$ AND Q_2 ... AND Q_n where each Q_i is of the form t_i FILTER $R_1^i \dots R_m^i$ with t_i a triple pattern and the R_j^i simple filter constraints. It is easy to see that P_2' can be computed in time polynomial in the size of P_2 and is itself of size at most $|P_2|^2$. Let $R_i = [\![Q_i]\!]_D$, for each $1 \leq i \leq n$. It is easy to see that each R_i can be computed in time polynomial in D and is of size at most $|D|$. Moreover, all mappings μ in R_i have $dom(\mu) = vars(t_i)$. In other words, every R_i can be seen as a relational table with relational schema $vars(t_i)$. Clearly (1) holds iff

$$\mu[X_1 \cap X_2] \notin \pi_{X_1 \cap X_2}(R_1 \bowtie \cdots \bowtie R_n), \qquad (2)$$

Since every R_i can be seen as a relational table, checking (2) is actually equivalent to answering a select-project-join (SPJ) relational algebra expression. Because P_2 is acyclic, so is the SPJ expression $\pi_{X_1 \cap X_2}(R_1 \bowtie \cdots \bowtie R_n)$ in the sense of relational database theory. From known results on answering acyclic SPJ queries [2, 6], it hence follows that condition (2) can be checked in time polynomial in $|P_2'|$, and $\sum_{i=1}^{n} |R_i|$. Since $|P_2'| \leq |P_2|^2$; since $\sum_{i=1}^{n} |R_i| \leq n|D|$; and since every step in the reasoning only involves polynomial time computations, (1) can hence be checked in time polynomial in the size of P_2, D, and μ. □

Conjunctive acyclic patterns. We finish this subsection with a few final observations on the CPF patterns present in the log. From Table 2 we obtain that in total 283859 / 159025 CPF patterns use FILTER. Of those, almost all (264703 or 93.25% / 152859 or 96.13%) use only simple filter constraints. Moreover, almost all CPF patterns (this includes also patterns in which FILTER does not occur), are acyclic (in particular: 2004486 or 99.99%/ 665652 or 99.99%).

This may be of interest to practical SPARQL processing engines, since known results from relational database theory imply that for acyclic CPF patterns with simple constraints the output $[\![P]\!]_D$ can be computed in time polynomial in the combined size of the input and the output, P, D and $[\![P]\!]_D$ [2] whereas this is not the case in general.

Acknowledgments. The authors are grateful to Sören Auer and Patrick van Kleef for sharing the DBPedia query log. The authors are also grateful to George Fletcher and Jan Hidders for inspirational discussions.

5. REFERENCES

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of VLDB 2007*, pages 411–422. ACM, 2007.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[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] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *Proceedings of VLDB 2005*, pages 1216–1227, 2005.

[5] G. H. L. Fletcher and P. W. Beck. Scalable indexing of RDF graphs for efficient join processing. In D. W.-L. Cheung, I.-Y. Song, W. W. Chu, X. Hu, and J. J. Lin, editors, *CIKM*, pages 1513–1516. ACM, 2009.

[6] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.

[7] 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*, 2010.

[8] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD 2009 Conference Proceedings*, pages 627–640, 2009.

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

[10] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[11] A. Polleres. From SPARQL to rules (and back). In *WWW 2007 Conference Proceedings*, pages 787–796.

[12] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Technical report, W3C Recommendation, 2008.

[13] Resource description framework((RDF). Technical report, W3C. `http://www.w3.org/RDF/`.

[14] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT 2010 Proceedings*, pages 4–33, 2010.

[15] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[16] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146. ACM, 1982.

[17] M.-E. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently joining group patterns in SPARQL queries. In *7th Extended Semantic Web Conference*, volume 6088 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2010.

[18] W3C SWEO Community Project. Linking open data. `http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`.

[19] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.