How is your Knowledge Graph Used: Content-Centric Analysis of SPARQL Query Logs*

Luigi Asprino[1] [0000−0003−1907−0677] and Miguel Ceriani[2,3] [0000−0002−5074−2112]

[1] University of Bologna, Via Zamboni 33, Bologna, Italy
[2] University of Bari Aldo Moro, Via Orabona 4, Bari, Italy
[3] ISTC-CNR, Via S. Martino della Battaglia 44, Roma, Italy
`luigi.asprino@unibo.it`
`miguel.ceriani@uniba.it`

Abstract. Knowledge graphs (KGs) are used to integrate and persist information useful to organisations, communities, or the general public. It is essential to understand how KGs are used so as to evaluate the strengths and shortcomings of semantic web standards, data modelling choices formalised in ontologies, deployment settings of triple stores etc. One source of information on the usage of the KGs is the query logs, but making sense of hundreds of thousands of log entries is not trivial. Previous works that studied available logs from public SPARQL endpoints mainly focused on the general syntactic properties of the queries disregarding the semantics and their intent. We introduce a novel, content-centric, approach that we call *query log summarisation*, in which we group the queries that can be derived from some common pattern. The type of patterns considered in this work is *query templates*, i.e. common blueprints from which multiple queries can be generated by the replacement of parameters with constants. Moreover, we present an algorithm able to summarise a query log as a list of templates whose time and space complexity is linear with respect to the size of the input (number and dimension of queries). We experimented with the algorithm on the query logs of the Linked SPARQL Queries dataset showing promising results.

Keywords: SPARQL · Query log summarisation · Linked SPARQL queries.

1 Introduction

Knowledge Graphs (KGs) are pervasive assets used by organisations and communities to share information with other stakeholders. For knowledge engineers, it is essential to understand how KGs are used so as to assess their strengths and shortcomings, but, neither established methodologies nor tools are available. We observe that it is customary to make KGs accessible via SPARQL endpoints,

* This is the extended version of [6].

therefore their query logs, i.e. the list of queries evaluated by the endpoint, are a valuable source from which the use of the KGs can be pictured. Compared to logs of "traditional" (centralised) databases (both relational and NoSQL), logs of public SPARQL endpoints bear much more information because they show usage of a dataset by multiple agents (human or robotic), for multiple applications, in different ways, and even in the context of multiple domains (especially if the dataset is generic).

Several works had already analysed the available SPARQL logs [30,2,33,17,38,12,11,39,10]. Most of them centred the analysis on the general structure of the queries (usage of specific SPARQL clauses, the shape of the basic graph patterns). The output of the analyses is mostly quantitative, possibly coupled by some examples. Relatively less focus has been so far given to aspects that go beyond the general query syntactic structure and relate to the actual content, such as aspects ranging from the usage of specific RDF terms (both classes, properties, and individuals), to specific (sub)query patterns, to inference of template usage and query evolution. Analysis of the actual content of queries can lead to further quantitative results, but most importantly can be used as a tool for qualitative analysis of one or multiple query logs: different levels of abstractions on the queries enable a meaningful exploration of the given data set.

The potential usage contexts for such analysis are manifold. For example, maintainers of SPARQL endpoints could optimise the execution of common queries by caching results or indexing predicates; designers of ontologies could assess what predicates are actually used thus allowing reshaping the model with shortcuts or removing unused predicates; designers of semantic web standards could introduce new constructs and operators in order to address common query patterns; and, researchers of the field could design benchmark to assess the performance of SPARQL endpoints.

The present work introduces a novel general approach to analyse query logs with a focus on query content and qualitative information. Specifically, we frame the *query log summarisation* as the problem of finding a list of templates modelling a query log. We introduce an algorithm able to solve the problem whose time and space complexity is linear in the size of the input. Finally, we experiment with the algorithm on the logs available in the LSQ dataset [38] to evaluate its usefulness. The analysis of the results shows that the method is able to provide more concise representations of the logs and novel insights on the usage of 28 public SPARQL endpoints.

The rest of the paper is organised as follows. Section 2 gives an overview of the existing work on query logs analysis. Section 3 lays the theoretical foundation of the work and introduces the problem of query log summarisation. The proposed algorithm to address the problem is presented in Section 4. Section 5 describes the experimental evaluation and its results, discussing strengths and opportunities enabled by the proposed approach. Section 6 concludes and outlines the ongoing and future work.

2 Related Work

Query logs are insightful sources for profiling the access to datasets. Although there are no approaches that aim to summarise SPARQL query logs as a list of query templates, an overview of the main approaches to analysing query logs is worthwhile. We classify the approaches according to the target query language.

Approaches targeting SQL query logs. Even if not directly applicable to assess the usage of knowledge graphs, techniques analysing query logs of relational databases may be adapted as SQL and SPARQL have syntactic similarities. These techniques have been used for detecting anomalous access patterns [22], preventing insider attacks [27] and optimising the workload of database management systems [15] thus becoming standard features for automatic indexing in commercial relational databases [29,32]. All the approaches can be generalised as feature extraction methods needed for clustering queries and profiling user behaviour. In most cases, the features extracted are basic, such as the SQL command used (e.g. SELECT, INSERT), the list of relations queried, and the operators used. Nevertheless, similarly to our approach, query templates and structural features are also used for computing query similarity [23,45], albeit still in a clustering approach. Some issues of such feature-based clustering approaches are that finding a useful way to convey the meaning of the clusters is not trivial, that scalability can be a problem as the worst-case cost is quadratic, and that some aspects of the query are scraped since the beginning for performance reasons, while they may be a relevant facet of a common pattern. Specifically, some methods [23,45,44] replace all the constants in the query with placeholders as a pre-processing step, which for SPARQL would hide the intent of most of the queries. Our method also replaces the constants with placeholders in an initial phase but, crucially, keeps the mapping with the original constants and puts them back if they have always the same value in a group of queries.

Approaches targeting SPARQL query logs. Analyses of SPARQL query logs have been performed since the early years of the Semantic Web. These studies fall into a more general line of research adopting empirical methods for observing typical characteristics of data [4,7], identifying common patterns in data [5], assessing the usage and identifying shortcomings of data [25,3] and using the obtained insights for developing better tools [21]. This kind of analysis has been also promoted by international workshops, such as USEWOD[4] which from 2011 to 2016 fostered research on mining the usage of the Web of Data [26]. Most of the existing work focus on quantitative and syntactic characteristics, such as the types of clients requesting semantic data [30] (including analyses of the characteristics of queries issues by humans, called organic, and those sent by artificial agents, robotic queries [10,37]), the user profile [20], the number of triple patterns per query [30,2,33,42,17], the use of predicates [30,2,33], the use of SPARQL operators [2,33,42,17] or a specific function (e.g. REGEX [1]), the

[4] `http://usewod.org/workshops.html`

structure of the Basic Graph Patterns (e.g. the out-degree of nodes, the number of join vertices) [2,42], the monotonicity of the queries [17], the probabilistic safeness [39], and the presence of non-conjunctive queries [33]. However, the analysis is limited at the triple-pattern level by paying less attention to the structural and semantic characteristics of the queries, thus making it difficult to figure out what the prototypical queries submitted to the endpoints look like. A noteworthy exception is [36], in which the author, while analysing queries at the triple pattern level, attempts to extract generic query patterns.

Bonifati et al. [12] investigate the structural characteristics related to the graph and hypergraph representation of queries by outlining the most common shapes. Moreover, they analyse the evolution of queries over time, by introducing the notion of the streak, i.e., a sequence of queries that appear as subsequent modifications of a seed query. By grouping queries based on similarity, this aspect of their work is akin to the approach presented in this work.

The existing studies are valuable for assessing the usage of SPARQL as a standard query language or for benchmarking and optimising the query engines. However, none of the existing approaches provides any insight into how KG is actually queried in terms of KG patterns queried by the users, and, therefore are of little help in designing the KGs. This paper investigates an alternative approach aiming at extracting query templates from SPARQL logs that may help designers to characterise the prototypical queries submitted by the users.

3 Preliminaries

This Section lays the theoretical foundation of this work.

RDF and SPARQL. For the sake of completeness, we introduce the basic notions of RDF [14] and SPARQL [18] needed to understand the methods and analysis described in this work. We defer the reader to the corresponding documentation for a complete description of these standards. Formally, let I, B, and L be infinite sets of IRIs, blank nodes, and literals. The sets are assumed to be pairwise disjoint and we will collectively refer to them as *RDF terms*. A tuple $(s, p, o) \in (I \cup B) \times (I) \times (I \cup B \cup L)$ is called *(RDF) triple* and we say s is the *subject* of the triple, p the *predicate*, and o the *object*. An *RDF graph* is a set of RDF triples, whereas an *RDF dataset* is a collection of named RDF graphs, each one identified by IRI, and a default RDF graph.

SPARQL is based on the idea of defining patterns to be matched against an input RDF dataset. Formally, considering the set of variables V, disjoint from the previously defined I, B, and L, a *triple pattern* is a tuple of the form $(s, p, o) \in (I \cup B \times V) \times (I \times V) \times (I \cup B \cup L \times V)$. A *basic graph pattern (BGP)* is a set of triple patterns. A SPARQL query Q is composed of the following components: *(i)* the query type (i.e. SELECT, ASK, DESCRIBE, CONSTRUCT); *(ii)* the dataset clause; *(iii)* the graph pattern (recursively defined as being a BGP or the result of the composition of one or more graph patterns through one of several SPARQL operators that modify and combine the obtained results); *(iv)* the solution modifiers (i.e. LIMIT, GROUP BY, OFFSET).

3.1 Query templates

Intuitively, a query template is a SPARQL query containing a set of placeholders which are meant to be substituted with RDF terms. The placeholders are called *parameters* of the query template and will be represented in queries using variable names starting with "`$_`"[5]. For example, consider the following queries 1.1 and 1.2[6]. The intent of both queries is to retrieve the types of a given entity. Such intent can be expressed via the Template 1.1.

Query 1.1:

```
SELECT ?type WHERE {
 dbr:Barack_Obama rdf:type ?type
}
```

Query 1.2:

```
SELECT ?type WHERE {
 dbr:Interstellar_(film) rdf:type ?type
}
```

Template 1.1:

```
SELECT ?type WHERE {
  $_1 rdf:type ?type
}
```

Query 1.3:

```
SELECT ?p WHERE {
 ?p rdf:type foaf:Person
}
```

Template 1.2:

```
SELECT ?type WHERE {
 $_1 $_2 ?type
}
```

We say that a query template q^t *models* a query q, indicated as $q^t \prec q$, if there exists a partial bijective function m^t, called *mapping*, that maps parameters P^t in q^t onto RDF terms of q such that applying m^t onto q^t gives q, i.e. $m^t : P^t \rightarrow (I \cup L)$ and $m(q^t) = q$. For example, the following mappings m_1 and m_2 transform the Template 1.1 into the queries 1.1 and 1.2 respectively: m_1`($_1) := dbr:Barack_Obama` and m_2`($_1) := dbr:Interstellar_(film)`.

It is worth noticing that, to preserve the intent of the query, templates do not substitute variables and blank nodes (as they are considered non-distinguished variables) with parameters, reduce the number of triple patterns, or replace SPARQL operators. As a result, a template for modelling a set of queries does not always exist (e.g. a single template modelling queries 1.1, 1.2, and 1.3 can not exist). Moreover, multiple templates may model the same set of queries. For example, the Template 1.2 models the queries 1.1 and 1.2 (in this case m_1 and m_2 must also map `$_2` onto `rdf:type`, i.e. m_1`($_2) := rdf:type` and m_2`($_2) := rdf:type`). In fact, the number of parameters of a template allows us to formalise the intuition of more specific/generic template. We say that the Template 1.2 is more generic (or, less specific) of Template 1.1 as it maps a higher number of parameters. As a result, given a query q, the *most generic*

[5] Using the initial underscore in the variable name to identify parameters matches with existing practice [28], while using "$" visually helps distinguish the parameters from query variables that often start with "?"

[6] For brevity, the queries omit prefix declarations:

- `dbr: <http://dbpedia.org/resource/>`
- `rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
- `foaf: <http://xmlns.com/foaf/0.1/>`
- `dbo: <https://dbpedia.org/ontology/>`

template modelling q is the template in which all the IRIs and literals of q are substituted by parameters. Therefore, it is easy to see that given a set of queries (that can be modelled by a single template) it is always possible to derive the most generic template by substituting all literals and IRIs by parameters.

We characterised templates as queries having placeholders that are to be replaced by IRIs or literals. However, there are two extensions to this rule which are needed to capture very common patterns for paginating the results and injecting values into a query. One is the usage of placeholders in `LIMIT` and `OFFSET` clauses, which are the solution modifiers used to get a specific slice of all the results. Both clauses are always followed by an integer, specifying respectively the number and initial position of the query solutions. By allowing this, integers to be replaced by parameters, multiple versions of the same query in which only one or both are changed (e.g., changing the `OFFSET` to perform pagination) can be represented by the same template.

The second extension to the rule has been defined for another specific clause: `VALUES`. This clause is used to bind one or more variables with a multiset of RDF terms. It is thus a way to give constraints to a query with multi-valued data that could come from previous computations, possibly also other queries[7]. In the case of a `VALUES` clause, rather than replacing single RDF terms, a placeholder either replace the whole corresponding multiset of terms or none.

Even if they are not explicitly mentioned, all the SPARQL clauses and operators (FILTERs, OPTIONALs, UNIONs etc.) can be part of a query template. We only mentioned the VALUE, LIMIT, and OFFSET operators as they deserve special treatment.

One of the main intuitions behind the usage of query templates to study a log is that it can help to "reverse engineer" the methods and processes used to generate the queries. In order to discuss this aspect, we define a *query-source* as a specific and unique piece of code (which could nevertheless span multiple software components in complex cases) that is responsible for the generation (possibly based on parameters) and execution of a query. A template that models many queries in a log may capture a common usage pattern that spans multiple query sources or a broadly used single query source. Both cases can be of interest in the analysis of a query log.

3.2 Query log summarisation problem

We formally describe a query log and frame its summarisation as a theoretical problem. A SPARQL Query Log $l = [e_1, e_2, .., e_n]$ is a list of entries $e_i = (q, t, s, m)$ each representing the execution at a certain time t of a query q by a SPARQL endpoint s with associated metadata m. For the purpose of the algorithm presented below, the information of the SPARQL endpoint executing the query is only used to group together queries evaluated over the same KG, and we do not consider time and metadata. Therefore, for brevity, SPARQL query logs reduce to a sequence of queries $l = [q_1, q_2, .., q_n]$. Note that queries can be

[7] It is for example a recommended way to perform query federation [35].

repeated in a log, so for convenience, we define an operator Q to get them as a set (hence without repetitions): $Q(l) = \{q_i | q_i \in l\}$.

Given a query log $l = [q_1, q_2 .., q_n]$, the *SPARQL log summarisation* is the problem of finding a set of query templates $Q_t = \{q_1^t, q_2^t, .., q_m^t\}$ (with $m \leq n$), called *log model*, such that for each query $q_i \in l$, there exists a query template q_j^t such that $q_j^t \prec q_i$. It is worth noticing that, since each query is the template of itself (in this case the mapping from placeholders to RDF terms is empty), a trivial solution to the problem is $Q_t = Q(l)$. Therefore, we have that the size of log model Q_t may range from 1, in the case that all the queries in the log are modelled by a single template, to $|Q(l)|$, when a common template for any pair of queries does not exist.

It is worth noticing that summarising a query log differs from evaluating the containment/equivalence of a pair of queries [13,34]. In fact, given a query q and its template q^t (i.e. $q^t \prec q$), q and q^t are (except for constants and parameters) the exact same query. Whereas evaluating the query containment/equivalence requires deciding if the result set of one query is always (i.e. for any dataset) contained into/equivalent to the result set of a *different* query. Of course, the two approaches, log summarisation and query containment/equivalence can be potentially combined to derive more succinct log models, but this is outside the scope of this paper.

Metrics. The aim of summarising a query log is to assist KG engineers in understanding how their KGs are queried. To do so, a KG engineer has ideally to go through the list of all the queries. Obviously, the shorter the list of queries to examine, the less effort from the KG engineer is required for the analysis. Intuitively, the benefit of using a log model instead of a full query log is to reduce the list of queries to examine. This benefit is proportional to the difference between the size of the log model and the size of the query log. However, one must consider that not all the query templates have the same informational value. In fact, we can consider that the more log entries a template models, the more informative it is (in other words, it allows the KG engineer to have an indication of a larger portion of the query log). Therefore, if the templates are ordered according to their informational value, the KG engineer would be able to analyse a large portion of the log by going only through the most informative templates.

To measure the impact of this on the informational value of a model we employ the concept of *entropy.* The entropy over a discrete random variable X, taking values in the alphabet $\mathcal{X}$ and distributed according to $p : \mathcal{X} \to [0, 1]$, is defined as follows [40]:

$$\mathrm{H}(X) := -\sum_{x \in \mathcal{X}} p(x) \log p(x)$$

Given a query log l and a model Q_t over it, we consider a random variable T taking values over the "alphabet" Q_t and distributed as the templates of Q_t are distributed over the log l. That is, with probability distribution p_{Q_t} defined as follows:

$$p_{Q_t}(q_i^t) = \frac{|\{q_j | q_j \in l, q_i^t \prec q_j\}|}{|l|}$$

We can thus measure the entropy of this distribution, which depends both on the log l and the model Q_t. The entropy corresponds to the average number of bits (considering base 2 for log) used to encode an item, which in our case is a template, in an optimal encoding. For a uniform distribution over n values, the entropy is $log(n)$, which is the number of bits required for a simple encoding of n values. If the values are not uniformly distributed a more efficient representation (as in a lossless compression) can be used, where more frequent values are represented with shorter encodings.

Recalling that the set of queries $Q(l)$ is already a model of l, the one created by simply taking all the queries as they are, we can compute the entropy for this model. The aim of another computed model Q_t of l is to achieve a more concise representation of the log and thus lower entropy. In the experiments with a dataset of logs (cf. Section 5), we measure the entropy of $Q(l)$ (indicated as $H(Q)$) as opposed to that of a derived log model Q_t (indicated as $H(T)$). The difference between $H(Q)$ and $H(T)$ indicates how much less information needs to be screened by the KG engineer to examine the log.

4 Approach

We describe the procedure for query log summarisation. Appendix A contains the complete pseudo-code for the algorithm, the sketch of the proof of soundness, the detailed complexity analysis, and other formal considerations on the output of the algorithm. To convey the intuition, we use the following log as a running example l =[Query 1.1, Query 1.2, Query 1.1, Query 1.4, Query 1.2, Query 1.5] where Queries 1.1 and 1.2 are defined above and Queries 1.4 and 1.5 follow.

Query 1.4:

```
SELECT ?director ?starring WHERE {
   dbr:Pulp_Fiction dbo:director ?
       director .
   dbr:Pulp_Fiction dbo:starring ?
       starring .
}
```

Query 1.5:

```
SELECT ?director ?starring WHERE {
   dbr:Django_Unchained dbo:director
       ?director .
   dbr:Django_Unchained dbo:starring
       ?starring .
}
```

Template 1.3:

```
SELECT ?director ?starring WHERE {
   $_1 $_2  ?director .
   $_3 $_4  ?starring .
}
```

Template 1.4:

```
SELECT ?director ?starring WHERE {
   $_1 dbo:director  ?director .
   $_1 dbo:starring ?starring .
}
```

Intuitively, the algorithm performs two steps, called GENERALISE() and SPECIALISE(). The function GENERALISE() creates a generic template for a query, replacing each occurrence of IRIs and literals with a different new parameter. Therefore, the generated template is the most generic that models the query. At the same time a mapping is created, associating each parameter with the RDF term that was replaced. For example, GENERALISE(Query 1.1) returns the Template 1.2 and the mapping m_1 defined as follows: m_1(**$_1**) := `dbr:Barack_Obama`, m_1(**$_2**) := `rdf:type`; GENERALISE(Query 1.2) returns the Template 1.2 and the mapping m_2 defined as follows: m_2(**$_1**) := `dbr:Interstellar_(film)`, m_2(**$_2**) := `rdf:type`; GENERALISE(Query 1.4) returns the Template 1.3 and the mapping m_4 defined as follows: m_4(**$_1**) := `dbr:Pulp_Fiction`, m_4(**$_2**) := `dbo:director`, m_4(**$_3**) := `dbr:Pulp_Fiction`, m_4(**$_4**) := `dbo:starring`; GENERALISE(Query 1.5) returns the Template 1.3 and the mapping m_5 defined as follows: m_5(**$_1**) := `dbr:Django_Unchained`, m_5(**$_2**) := `dbo:director`, m_5(**$_3**) := `dbr:Django_Unchained`, m_5(**$_4**) := `dbo:starring`.

The function SPECIALISE() takes as input a template and an associated set of mappings and, by just analysing the set of mappings, it establishes if the number of parameters can be reduced. There are two interesting cases for this purpose: *(i)* for a parameter, all the mappings in the set map it to the same RDF term (it is thus a constant); *(ii)* for a pair of parameters of a template, each mappings in the set maps them to a common RDF term (one parameter is actually a duplicate of the other). For each instance of these cases, the template and the mappings are updated accordingly: *(i)* in the first case (the parameter is constant), the parameter in the template is replaced by the constant and removed from the mappings; *(ii)* in the second case (two parameters mapped to the same RDF terms), one parameter in the template is replaced by the other and removed from the mappings. For example, both m_1 and m_2 map **$_2** to `rdf:type` which can be considered as a constant (i.e. m_1(**$_2**) = m_2(**$_2**) = `rdf:type`), therefore the Template 1.2 can be specialised as Template 1.1 and the parameter **$_2** replaced with `rdf:type`. Concerning the Template 1.3 and the mappings m_4 and m_5, the SPECIALISE function replaces **$_2** and **$_4** with two constants (`dbo:director` and `dbo:starring`) and unifies **$_1** and **$_3** in both mappings as they map to the same RDF term (`dbr:Pulp_Fiction` and `dbr:Django_Unchained` respectively for m_4 and m_5). The function returns the Template 1.4 and m_4 and m_5 updated.

The main function DISCOVERTEMPLATES(): *(i)* takes a set of queries; *(ii)* extracts a pair (template, mapping) for each query by invoking GENERALISE; *(iii)* accumulates the mappings associated with the same template into a dictionary (the dictionary uses the templates as keys and mapping sets as values); *(iv)* then, for each pair (template, mapping set), calls SPECIALISE() and, possibly, replaces the pair with a specialised one.

Furthermore, along with the mappings, the algorithm maintains the original query ids, which in turn allows to find the data of each corresponding execution in the log. Keeping track of this relationship is crucial so that is later possible to derive statistics based on their usage or explore the detail of specific executions.

Properties of the extracted log model. It is worth noticing that, given a query log, the algorithm first maximizes the number of queries a single template can represent, by grouping each query under its most generic template. Then, the algorithm minimizes the number of parameters of each template, by returning the most specific template modelling that group of queries (in other words, it keeps a minimal set of parameters needed to represent the set of queries). This ensures that for any pair queries of the log, if a single template can model the queries, then, the template is in the log model and the template is the most specific one.

Moreover, since the algorithm does not perform any normalisation of the input queries, syntactic differences affect the templates, e.g. two queries having the same triple patterns in a different order result in two different templates. This implies that the extracted templates generalise over fewer input queries (hence the algorithm tends to extract more templates) in respect to what could be if some normalisation was adopted, but the extracted templates are closer to the queries sent by the clients (which is desirable for identifying queries sent from the same process). Some form query normalisation can then be included as a preliminary step for different perspectives, but this is left to future work.

Implementation of the algorithm. The algorithm has been implemented in Javascript, relying on the SPARQL.js library[8] for SPARQL parsing. Both the LSQ dataset in input and the discovered templates are represented as RDF in a local triple store, namely Apache Jena Fuseki[9]. The code is freely available on GitHub[10]

5 Experimentation

The LSQ dataset, already briefly introduced in Section 2, is the de-facto state-of-the-art collection of SPARQL query logs. We tested our method by using it to analyse all the logs available in the latest version of the LSQ dataset. In this section, we describe and discuss the dataset, its analysis, and the findings, focusing on the high level view and the details that can be useful to discuss the algorithm. For the detailed description of the results obtained for each endpoint and the full code of all the templates we refer the reader respectively to Appendix B and C.

5.1 The Dataset

The LSQ 2.0 dataset[11] contains information about approximately 46M query executions and is composed of logs extracted from 28 public SPARQL endpoints. 24 of the endpoints are part of **Bio2RDF**, a project aimed at converting to RDF different collections of heterogeneously formatted structured biomedical data [9]. The other four endpoints are the following ones: **DBpedia**, a well-known knowledge base automatically extracted from Wikipedia [8]; **Wikidata**,

[8] `https://github.com/RubenVerborgh/SPARQL.js`
[9] `https://jena.apache.org/documentation/fuseki2`
[10] `https://github.com/miguel76/sparql-clustering`
[11] `http://lsq.aksw.org/`

an encyclopedic knowledge graph built collaboratively [43]; **Semantic Web Dog Food (SWDF)**, a dataset describing research in the area of the semantic web [31]; **LinkedGeoData** [41], an RDF mapping of OpenStreetMap, which is, in turn, a user-curated geographical knowledge base [16].

The LSQ project provides the collection of these SPARQL logs and their conversion to a common (RDF-based) format. In the process of conversion, the LSQ software performs also some filtering (e.g., only successful queries are considered) and anonymisation (e.g., client host information is hidden). The main information items offered by LSQ from each entry of a query log are the following ones: the endpoint against which the query was executed; the actual *SPARQL query*, the *timestamp* of execution, and an anonymised identifier of the client *host* which sent the query.

Dataset	**Execs**	**Hosts**	**Queries**	**H(Q)**	**Templ.s**	**H(T)**	**ΔH**
Bio2RDF	**33 829 184**	**2 306**	**1 899 027**	**15.22**	**12 296**	**3.73**	**11.49**
DBpedia	**6 999 815**	**37 056**	**4 257 903**	**21.16**	**17 715**	**5.58**	**15.59**
DBpedia-2010	518 717	1 649	358 955	17.99	2 223	5.66	12.33
DBpedia-2015/6	6 481 098	35 407	3 903 734	21.01	15 808	5.21	15.80
Wikidata	**3 298 254**	**-**	**844 260**	**12.26**	**167 578**	**7.47**	**4.80**
LinkedGeoData	**501 197**	**25 431**	**173 043**	**14.24**	**2 748**	**4.78**	**9.46**
SWDF	**1 415 568**	**921**	**101 422**	**14.54**	**1 826**	**1.03**	**13.51**

Table 1: Statistics on the LSQ 2.0 dataset before/after summarisation.

Table 1 shows some statistics about the data in the LSQ dataset, organised by endpoints[12]. The column *Execs* indicates the number of query executions contained in the log. Column *Hosts* is the total number of client hosts and *Queries* is the number of unique queries. The column H(Q) is the entropy of the unique queries distribution across the executions.

5.2 Methodology of Analysis

The aforementioned templates-mining algorithm was applied separately on each query log in the LSQ 2.0 dataset, with the corresponding set of queries as input. Furthermore, the queries of Bio2RDF were also considered as a whole, on top of analysing each specific endpoint[13]

The templates obtained with our method can be analysed in a variety of ways. Different statistics can be computed on top of this summarised representation

[12] In the table, for conciseness, the statistics of the Bio2RDF endpoints are shown only aggregated for the whole project. In Appendix B there is a more detailed version of the table showing the statistics endpoint by endpoint.

[13] This choice is motivated by the fact that the Bio2RDF endpoints are part of the same project, the collected logs refer roughly to the same period, and there is considerable overlap in the clients querying the endpoints.

of the original data. Furthermore, the templates can be explored in several ways to have a content-based insight of how an endpoint has been used. In this study we will focus on two main aspects:

- a quantitative analysis of the effectiveness of the summarisation by measuring for each log 1) the number of templates in comparison with the number of queries and 2) the entropy of the templates distribution in comparison with the entropy of the query distribution;
- a qualitative analysis of the templates obtained, choosing for each log the ten most executed ones and discussing the possible intent of the queries, what they say about the usage of the endpoint, which ones probably come from a single code source, which ones instead probably correspond to common usage patterns, if and how some of them are related between each other.

It should be noted many other perspectives are possible (some of them will be sketched among the future work in Section 6).

5.3 Results

The execution of the algorithm overall took approximately nine hours on consumer hardware. Statistics about the results for each log or set of logs are shown in Table 1, alongside the previously described information. The column *Templ.s* corresponds to the number of templates generated, while the column $\mathrm{H}(T)$ is the entropy of the templates distribution across the log and $\Delta\mathrm{H}$ is the difference between the entropy according to the unique queries and the one according to the templates ($\Delta\mathrm{H} = \mathrm{H}(Q) - \mathrm{H}(T)$).

For all the logs the number of templates is significantly smaller than the number of unique queries, with a reduction amounting to around two orders of magnitude (the ratio going from ∼56 to ∼240) for all cases but Wikidata (for which the reduction is smaller, namely five-fold). The reduction in entropy considering the distribution using templates shows even more strongly the effectiveness of the summarisation, as the value is in all the cases greater than $log_2 \frac{|Q|}{|T|}$, which would be the reduction in entropy in case of uniform distributions, showing that the algorithm is able to merge the most relevant (in terms of executions) queries.

Furthermore, it is worth noticing that, regarding the DBpedia log, while there is a significant difference in the query entropy from the data of 2010 (17.99) to the ones of 2015/6 (21.01), in line with a ten-fold increase in both executions and unique queries, the respective entropies measured on templates distribution are much closer, actually sightly decreasing from 2010 (5.66) to 2015/6 (5.21). This is interesting because it shows that the template diversity remains stable, while the number and diversity of specific queries increase roughly as the volume of the executions. In our opinion this case also manifests the importance of using the entropy as an index of diversity, rather than just counting the total number templates (which is instead quite different between the two datasets, ∼2.2K against ∼16K).

Then, for each endpoint[14], we performed the qualitative analysis of the ten most frequently executed templates. As part of the interpretation of these templates, we labelled them using a functional syntax composed of the a name given to the function (template) and a name given to each parameter. Interestingly, the most executed templates are quite vary across different endpoints and fulfil different kinds of purposes. Some templates correspond to generic, content-independent, patterns, like the template from SWDF log labelled PROPERTIESANDVALUES(*resource*) that list all properties and values associated to a resource and has been executed ~17K times. Others are specific of some triple store software as they use specific extensions, as it is the case for as in the template COMMONSUPERCLASSANDDISTANCE(*class1*,*class2*) from Wikidata, executed ~107K times, which employs a feature specific of Blazegraph, the software used for this dataset. Others are specific of some domain that the dataset encompasses, like CLOSEPOIS(*latitude*,*longitude*) from LinkedGeoData, executed ~81K times, that looks for points of interest close to a geographic location. Some of them, finally, are specific of a certain application, like AIRPORTSFORCITY(*cityLabel*,*lang*) in DBpedia, executed ~1.4M times,.

As previously mentioned, it can be of interest to understand if a template correspond to a single query-source or instead arises from a pattern which is common in the usage of an endpoint. While we do not propose a specific metric for this purpose, nor we have a general way to check the ground truth, the qualitative analysis of the most executed templates offers a chance to reason on this topic. The generality of the template, as accessed above, offers a hint: the more general the more likely that it correspond to commonly adopted pattern rather than a single query-source. But the analysis of the general-purpose templates found show that they are not necessarily simple and may not correspond to the most straightforward solution to design a certain query. The structural complexity is perhaps then a better predictor of the usage of a template. For example, the template TRIPLES(*subject*) in Bio2RDF is a `construct` that return all the triples for which *subject* is the subject. The query is hence functionally generic but it is peculiar for being in a form slightly more complex than necessary: it is composed of a triple pattern and a `filter` instead of using directly a triple pattern with fixed subject. This template has been executed across most of the endpoints of Bio2RDF, for a total of ~9.3M times.

Another interesting aspect that emerges from the qualitative analysis is the evidence of relationships between different templates. For each endpoint, even considering just the most executed templates, it is possible to find one or more groups of templates that for structure, function, number of executions, hosts, period of use show many commonalities and can reasonably be conjectured to be part of a common process. For example among the most executed templates on SWDF four of them have been executed the same number of times and have the same kind of parameter (a researcher) albeit they extract different kind of data (respectively general information, affiliations, participation to events, publications). Still on SWDF, there are other two groups of templates having the

[14] With the exception of the Bio2RDF endpoints, which are considered as a whole.

same aspects in common (with a group having as common parameter an article and another having as common parameter an organisation). While in this case the grouped templates are probably part of a single process that executes multiple queries, in other cases the related templates could testify the evolution of a process. The template commonSubclasses(*class1*, *class2*) from the LinkedGeoData log is executed ∼17K times across a span of ∼7 hours, then it is "replaced" by the template commonSubclasses(*class1*, *class2*, *class3*) that fulfills the same purpose but having one class more as parameter. The second version is then executed ∼17K times across a span of other ∼7 hours.

Such hypothesises about the relationship between among a group of queries are reinforced in all the cases we found by the fact that the templates are executed by a common set of hosts. In most of the cases it is a single host that execute all the templates in a group, but not necessarily: on DBpedia the templates countLinksBetween(*res1*, *res2*) and countCommonLinks(*res1*, *res2*) have different but related functions[15] on the same kind of parameters, they are both executed ∼181K times by the same set of ∼1130 hosts.

The complete results are available online for download[16] The templates found for each endpoint are represented both as CSVs and RDF. The RDF representation of the templates is meant to be used alongside the RDF representation of LSQ and is based on the Provenance Vocabulary [19], a specialisation of the standard W3C provenance ontology (PROV-O) [24] dealing with web data and in particular SPARQL queries and query templates.

5.4 Discussion

The aim of the analysis of the LSQ dataset was to prove that our method is able to effectively summarise the given logs, that the inferred templates often correspond to broadly used patterns or single query-sources, and that their analysis can give new insights on the usage of the considered endpoints. We quantitatively measured the efficacy of the summarisation through the ratio of original queries per template and the reduction in entropy when considering each log entry as an instance of a template, rather than as an instance of a query. Both measures show that the summarisation had a noteworthy impact on all the considered logs. Moreover, the qualitative analysis of a selected sample of templates (specifically the most executed) shows how their function may be appropriately analysed and discussed, without the need to check the thousands of corresponding queries.

Regarding the accuracy of the predicted templates in identifying a single source for a set of queries, there is no gold standard or previous attempt to compare with. Thus the qualitative analysis resorts to educated guesses, where we decide if an inferred template corresponds plausibly to a single source based on the syntactic distinctness and relationship with other templates and data from the log. For many of the described templates, it is possible to reasonably

[15] One counts the triples in which one resource is subject and the other object, the other counts the triples in which they replace each other or have symmetric role.

[16] `https://doi.org/10.6084/m9.figshare.23751138`

infer a single origin. In terms of the usefulness of the inferred templates to gain insights, the qualitative analysis has shown multiple ways in which the analysis of the templates gives direct access to information that was previously not straightforward and stimulates further study.

Finally, another finding has been that this template-based analysis paves the way to the analysis of another level of relationships between queries, namely when different queries are applied to the same (or related) data items as part of a (possibly automatic) process. Evidence of such relationships has been found in the qualitative analysis of all the considered logs.

6 Conclusions

In this work, we address the *query log summarisation* problem, i.e. identifying a set of *query templates* (i.e. queries with placeholder meant to be replaced with RDF terms) describing the queries of a log. We designed and implemented a method to perform the summarisation of a query log in linear time, based on the use of a hash table to group sets of queries that can be derived from a common query template. The approach has been experimented with the available logs of the LSQ dataset. The representation of the logs using templates has been shown to be significantly more concise. A qualitative analysis performed on the most executed templates enabled the characterisation of the log in ways that would not have been directly possible by analysing just the single queries.

Besides further exploring possible extensions of the template-mining algorithm for normalising the input log (e.g. reordering triple patterns), the analysis of the discovered templates brought forward some interesting issues that we consider deserving of further research.

One aspect worth investigating is the relationships between the execution patterns of each template. In the qualitative analysis, we found groups of templates being executed by the same set of hosts, often at similar times, and many times with the same parameters. Such analysis may, for example, allow to mine the prototypical interactions (namely, processes) with data, beyond the single query or template.

Moreover, many more interesting levels of abstraction are possible beyond the query templates: e.g., a common part of the query, the usage of certain BGP, a property, and so on. The general idea of the approach and the structure of the algorithm can be still applied. Apart from computing these multiple levels, which can be done by extending the presented algorithm, it is interesting to understand if some measure may be used to select the more relevant abstractions, rather than leaving the choice entirely to the user.

Another direction worth exploring is to assess the possible benefits of combining log summarisation with strategies for bot detection (e.g. templates can help characterise the features of queries and thus favouring the classification of robotic queries) or for optimising the execution of a sequence of queries (once prototypical interaction with data is delineated, one could imagine triple stores being able to predict workload and optimise query execution).

In this work, we mainly focussed on the most frequent queries, but, future analyses may also investigate what insights can be extracted from the rare ones (for example, a long tail of rare queries may indicate a high variety of clients and data exposed by the endpoint).

Finally, the proposed method and algorithm are applicable without much change to other query languages, thus offering an approach for the analysis of logs of, e.g. relational databases.

Supplemental Material Statement. The dataset with the experimentation results is publicly available (see note 16). The query logs used in the experimentation can be downloaded from the LSQ website[17]. The code is available from a public git repository (see note 10).

Acknowledgements This work was partially supported by the PNRR project "Fostering Open Science in Social Science Research (FOSSR)" (CUP B83C22003950001) and by the PNRR MUR project PE0000013-FAIR.

References

1. Aljaloud, S., Luczak-Rösch, M., Chown, T., Gibbins, N.: Get all, filter details-on the use of regular expressions in sparql queries. Proceedings of the Workshop on Usage Analysis and the Web of Data (USEWOD 2014) (2014)
2. Arias, M., Fernandez, J.D., Martinez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. Proceedings of Usage Analysis and the Web of Data (USEWOD 2011) (2011)
3. Asprino, L., Basile, V., Ciancarini, P., Presutti, V.: Empirical analysis of foundational distinctions in linked open data. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence and the 23rd European Conference on Artificial Intelligence (IJCAI-ECAI 2018). pp. 3962–3969 (2018). https://doi.org/10.24963/ijcai.2018/551
4. Asprino, L., Beek, W., Ciancarini, P., van Harmelen, F., Presutti, V.: Observing LOD using equivalent set graphs: It is mostly flat and sparsely linked. In: Proceedings of the 18th International Semantic Web Conference (ISWC 2019), Part I. pp. 57–74 (2019). https://doi.org/10.1007/978-3-030-30793-6_4
5. Asprino, L., Carriero, V.A., Presutti, V.: Extraction of common conceptual components from multiple ontologies. In: Proceedings of the International Conference on Knowledge Capture (K-CAP 2021). pp. 185–192 (2021). https://doi.org/10.1145/3460210.3493542
6. Asprino, L., Ceriani, M.: How is your knowledge graph used: Content-centric analysis of sparql query logs. In: The Semantic Web – ISWC 2023. pp. 197–215. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-47240-4_11
7. Asprino, L., Presutti, V.: Observing LOD: its knowledge domains and the varying behavior of ontologies across them. IEEE Access **11**, 21127–21143 (2023). https://doi.org/10.1109/ACCESS.2023.3250105

[17] http://lsq.aksw.org/

8. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: A Nucleus for a Web of Open Data. In: Proceedings of the International Semantic Web Conference (ISWC 2007). pp. 722–735. Springer (2007)
9. Belleau, F., Nolin, M.A., Tourigny, N., Rigault, P., Morissette, J.: Bio2rdf: towards a mashup to build bioinformatics knowledge systems. Journal of biomedical informatics **41**(5), 706–716 (2008)
10. Bielefeldt, A., Gonsior, J., Krötzsch, M.: Practical linked data access via SPARQL: the case of wikidata. In: Proceedings of the Workshop on Linked Data on the Web co-located with The Web Conference (LDOW@WWW 2018) (2018)
11. Bonifati, A., Martens, W., Timm, T.: Navigating the maze of wikidata query logs. In: Proceedings of The Web Conference (WWW 2019). pp. 127–138 (2019). `https://doi.org/10.1145/3308558.3313472`
12. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. VLDB Journal **29**(2-3), 655–679 (2020). `https://doi.org/10.1007/s00778-019-00558-9`
13. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: SPARQL query containment under SHI axioms. In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2012) (2012)
14. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax, `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`
15. Deep, S., Gruenheid, A., Koutris, P., Viglas, S., Naughton, J.F.: Comprehensive and efficient workload summarization. Datenbank-Spektrum **22**(3), 249–256 (2022). `https://doi.org/10.1007/s13222-022-00427-w`
16. Haklay, M., Weber, P.: Openstreetmap: User-generated street maps. IEEE Pervasive Computing **7**(4), 12–18 (2008). `https://doi.org/10.1109/MPRV.2008.80`
17. Han, X., Feng, Z., Zhang, X., Wang, X., Rao, G., Jiang, S.: On the statistical analysis of practical SPARQL queries. In: Proceedings of the 19th International Workshop on Web and Databases (2016). `https://doi.org/10.1145/2932194.2932196`
18. Harris, S., et al.: SPARQL 1.1 Query Language, `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/`
19. Hartig, O.: Provenance information in the web of data. In: Proceedings of the Workshop on Linked Data on the Web (LDOW 2009) (2009), `http://ceur-ws.org/Vol-538/ldow2009_paper18.pdf`
20. Hoxha, J., Junghans, M., Agarwal, S.: Enabling semantic analysis of user browsing patterns in the web of data. Proceedings of Usage Analysis and the Web of Data (USEWOD 2012) (2012)
21. Huelss, J., Paulheim, H.: What SPARQL query logs tell and do not tell about semantic relatedness in LOD - or: The unsuccessful attempt to improve the browsing experience of dbpedia by exploiting query logs. In: Proceedings of ESWC 2015, Revised Selected Papers. pp. 297–308 (2015). `https://doi.org/10.1007/978-3-319-25639-9_44`
22. Kamra, A., Terzi, E., Bertino, E.: Detecting anomalous access patterns in relational databases. VLDB Journal **17**(5), 1063–1077 (2008). `https://doi.org/10.1007/s00778-007-0051-4`
23. Kul, G., Luong, D., Xie, T., Coonan, P., Chandola, V., Kennedy, O., Upadhyaya, S.J.: Summarizing large query logs in ettu. CoRR (2016), `http://arxiv.org/abs/1608.01013`
24. Lebo, T., Sahoo, S., McGuinness, D.: PROV-O: The PROV Ontology, `https://www.w3.org/TR/2013/REC-prov-o-20130430/`

25. Luczak-Rösch, M., Bischoff, M.: Statistical analysis of web of data usage. In: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn2011) (2011)
26. Luczak-Rösch, M., Hollink, L., Berendt, B.: Current directions for usage analysis and the web of data: The diverse ecosystem of web of data access mechanisms. In: Proceedings of the 25th International Conference on World Wide Web (WWW 2016). pp. 885–887 (2016). https://doi.org/10.1145/2872518.2891068
27. Mathew, S., Petropoulos, M., Ngo, H.Q., Upadhyaya, S.J.: A data-centric approach to insider attack detection in database systems. In: Proceedings of the 13th International Symposium on Recent Advances in Intrusion (RAID 2010). pp. 382–401 (2010). https://doi.org/10.1007/978-3-642-15512-3_20
28. Meroño-Peñuela, A., Hoekstra, R.: grlc makes github taste like linked data apis. In: Proocedings of ESWC 2016. pp. 342–353 (2016)
29. Microsoft: Automatic Tuning - Microsoft SQL Server, https://learn.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning?view=sql-server-ver16
30. Möller, K., Hausenblas, M., Cyganiak, R., Handschuh, S.: Learning from linked open data usage: Patterns & metrics. In: Proceedings of the Web Science Conference (2010)
31. Möller, K., Heath, T., Handschuh, S., Domingue, J.: Recipes for semantic web dog food - the ESWC and ISWC metadata projects. In: Proceedings of the 6th International Semantic Web Conference and the 2nd Asian Semantic Web Conference, ISWC-ASWC 2007. pp. 802–815 (2007). https://doi.org/10.1007/978-3-540-76298-0_58
32. Oracle: Automatic Indexing - Oracle SQL Developer Web, https://docs.oracle.com/en/database/oracle/sql-developer-web/19.2.1/sdweb/automatic-indexing-page.html#GUID-8198E146-1D87-4541-8EC0-56ABBF52B438
33. Picalausa, F., Vansummeren, S.: What are real SPARQL queries like? In: Proceedings of the International Workshop on Semantic Web Information Management (SWIM 2011) (2011). https://doi.org/10.1145/1999299.1999306
34. Pichler, R., Skritek, S.: Containment and equivalence of well-designed SPARQL. In: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 14). pp. 39–50 (2014). https://doi.org/10.1145/2594538.2594542
35. Prud'hommeaux, E., Buil-Aranda, C.: SPARQL 1.1 Federated Query, http://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321/
36. Raghuveer, A.: Characterizing machine agent behavior through sparql query mining. In: Proceedings of the Workshop on Usage Analysis and the Web of Data (USEWOD 2012) (2012)
37. Rietveld, L., Hoekstra, R., et al.: Man vs. machine: Differences in sparql queries. In: Proceedings of the Workshop on Usage Analysis and the Web of Data (USEWOD 2014) (2014)
38. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.N.: LSQ: the linked SPARQL queries dataset. In: Proceedings of the 14th International Semantic Web Conference (ISWC 2015) Part II. pp. 261–269 (2015). https://doi.org/10.1007/978-3-319-25010-6_15
39. Schoenfisch, J., Stuckenschmidt, H.: Analyzing real-world SPARQL queries and ontology-based data access in the context of probabilistic data. International Journal of Approximate Reasoning **90**, 374–388 (2017). https://doi.org/10.1016/j.ijar.2017.08.005

40. Shannon, C.E.: A mathematical theory of communication. The Bell System Technical Journal **27**(3), 379–423 (1948). https://doi.org/10.1002/j.1538-7305.1948.tb01338.x
41. Stadler, C., Lehmann, J., Höffner, K., Auer, S.: Linkedgeodata: A core for a web of spatial open data. Semantic Web **3**(4), 333–354 (2012). https://doi.org/10.3233/SW-2011-0052
42. Stadler, C., Saleem, M., Mehmood, Q., Buil-Aranda, C., Dumontier, M., Hogan, A., Ngomo, A.C.N.: Lsq 2.0: A linked dataset of sparql query logs. (Preprint) (2022), https://aidanhogan.com/docs/lsq-sparql-logs.pdf
43. Vrandečić, D.: Wikidata: A new platform for collaborative data collection. In: Proceedings of the 21st International Conference on World Wide Web (WWW 2012). p. 1063–1064 (2012). https://doi.org/10.1145/2187980.2188242
44. Wang, J., Li, T., Wang, A., Liu, X., Chen, L., Chen, J., Liu, J., Wu, J., Li, F., Gao, Y.: Real-time Workload Pattern Analysis for Large-scale Cloud Databases. arXiv e-prints arXiv:2307.02626 (Jul 2023). https://doi.org/10.48550/arXiv.2307.02626
45. Xie, T., Chandola, V., Kennedy, O.: Query log compression for workload analytics. VLDB Endowment **12**(3), 183–196 (2018). https://doi.org/10.14778/3291264.3291265

Appendix A

Algorithm details

This appendix contains the complete pseudo-code for the algorithm (Algorithm 1) sketched in Section 4, the sketch of the proof of soundness and the detailed complexity analysis.

Proof of the soundness of the algorithm It is worth noticing that the only operation the algorithm performs on the queries and the templates is the replacement of RDF terms with parameters, and vice versa. Therefore, it is straightforward to see that for each query of the log, there is always a corresponding template.

Time and space complexity Given a log $l = [q_1, q_2, ..., q_n]$, the input to the algorithm is the set of the queries $Q(l)$. Let $|q_i|$ be the size of a query q_i (its length as string in SPARQL syntax) and $\text{SIZE}(Q(l))$ the size of the input, such that

$$\text{SIZE}(Q(l)) = \sum_{q_i \in Q(l)} |q_i|$$

We assume that the operations of storing and retrieving items from a dictionary can be done in $\Theta(1)$[18]. The time complexity of GENERALISE is $\Theta(|q_i|)$ as the main operation of the function is to parse the query, replacing each RDF term with a parameter. In the parent function DISCOVERTEMPLATES the function GENERALISE is called once for each query in $Q(l)$. The only other operation in the loop is to store the result in the dictionary *dict*, an operation that we assumed to cost $\Theta(1)$. The time cost of that loop as a whole is thus $\Theta(\text{SIZE}(Q(l)))$.

After the loop, *dict* will contain a set of templates $\{q_1^t, q_2^t, ..., q_m^t\}$. Let r_i be the number of parameters contained in the template q_i^t and n_i the number of mappings in the corresponding mapping set. The time complexity of SPECIALISE on a template q_i^t is $\Theta(r_i n_i)$. In fact, the first loop in SPECIALISE iterates over the parameters (r_i) and checks if all the mappings (n_i) map the parameter on the same constant. It hence costs $\Theta(r_i n_i)$. Then, for each pair of parameters, SPECIALISE checks if both parameters are always mapped to the same RDF term. A naive implementation of this loop would cost $\Theta(r_i^2 n_i)$, because every pair of parameters needs to be compared. But as we are just checking for equality, we can use a dictionary to perform the comparisons, so that each parameter can be compared with all the previously considered ones in one pass by checking the existence of an equal sequence of values in the dictionary. As again storing and retrieving items from a dictionary are assumed to be executed is $\Theta(1)$, the time complexity of this loop is $\Theta(r_i n_i)$ too. In the second loop of

[18] A time of $\Theta(1)$ in the average case can be achieved using a hash table.

Algorithm 1 Template mining algorithm

```
1: function DISCOVERTEMPLATES($querySet$)
2:     for all $query \in querySet$ do
3:         $(template, mapping) \leftarrow$GENERALISE$(query)$;
4:         if $template \notin dict$ then
5:             $dict[template] \leftarrow \{mapping\}$;
6:         else
7:             $dict[template] \leftarrow dict[template] \cup \{mapping\}$;
8:         end if
9:     end for
10:    $output \leftarrow \emptyset$;
11:    for all $template \in$ keys of $dict$ do
12:        $mappingSet \leftarrow dict[template]$;
13:        $(template', mappingSet') \leftarrow$
                        SPECIALISE$(template, mappingSet)$;
14:        $output \leftarrow output \cup (template', mappingSet')$;
15:    end for
16:    return $output$;
17: end function
18:
19: function GENERALISE($query$)
20:    $template \leftarrow query$;
21:    $mapping \leftarrow \{\}$;
22:    for all $RDFTerm \in query$ do
23:        $param \leftarrow$ create a new parameter;
24:        $template \leftarrow$ replace $RDFTerm$ with $param$ in $template$;
25:        $mapping[param] \leftarrow RDFTerm$;
26:    end for
27:    return $(template, mapping)$;
28: end function
29:
30: function SPECIALISE($template, mappingSet$)
31:    for all $p \in$ parameters of $template$ do
32:        if $\exists k$ s.t. $\forall m \in mappingSet.m[p] = k$ then
33:            $template \leftarrow$ replace $p$ with $k$ in $template$;
34:            $mappingSet \leftarrow mappingSet$ excluding $p$;
35:        end if
36:    end for
37:    for all $p, p' \in$ parameters of $template$ (with $p \neq p'$) do
38:        if $\forall m \in mappingSet, m[p] = m[p']$ then
39:            $template \leftarrow template$ with $p'$ replaced by $p$;
40:            $mappingSet \leftarrow mappingSet$ excluding $p$';
41:        end if
42:    end for
43:    return $(template, mappingSet)$;
44: end function
```

discoverTemplates, the function specialise is called once for each one of the templates $\{q_1^t, q_2^t, ..., q_m^t\}$, the global cost being thus $\Theta(\sum_{i=1}^{m} r_i n_i)$. Considering that $r_i < |q_i^t|$ and, given that the generalise do not increase the size of the query when it replaces the RDF terms, the cost can be written as $O(\sum_{i=1}^{m} \sum_j q_j)$, where q_j is every query modelled by the template q_i^t. Finally, as by construction each query is associated only to one template, the time complexity of the whole algorithm is $\Theta(\text{SIZE}(Q(l)))$, i.e. linear in respect of the input size.

It is worth noticing that the main data structure used by the algorithm is the dictionary for storing the templates, hence, the space complexity is $\Theta(\sum_{i=1}^{m} r_i n_i)$ that in the worst case (if no pair of queries can be modelled by a common template, ending up thus all in separate entries of the dictionary) is linear in the input size. In practice, experimentation shows that the number of templates is much smaller than the number of queries so the actual used space is much less.

Appendix B

Additional Experimental Details

This appendix extends the discussion of the experimentation presented in Section 5. Table 2 is a richer version of Table 1, in which statistics of the specific datasets composing Bio2RDF are shown. The following sections contain, for each dataset in LSQ, a thorough analysis of the available data and the obtained results.

1 Bio2RDF

Data from Bio2RDF endpoints consist of one log for each endpoint. The logs encompass data in the time interval from the 5th of May of 2013 to the 28th of September of 2014 [19].

The number of query executions (column *Execs* of Table 1) for each of the logs varies quite a lot, from 66K in the smallest log (KEGG) to 7.7M in the largest one (Taxonomy). The number of unique queries performed (column *Queries* of the table) is fairly smaller for all the Bio2RDF endpoints, from around three times smaller for BioModels (435K unique queries against 1.24M executions) to around 22 times smaller for Taxonomy (355K unique queries against 7.7M executions). As a whole, the Bio2RDF logs contain around 34M query executions but "only" around 1.9 M globally unique queries, with a considerable overlap of queries among endpoints. If the $\sim$1.9M queries were evenly distributed, the entropy of the queries (column $\mathrm{H}(Q)$) would have been $\log(\sim 1.9M) =\sim 20.86$. The value is 15.22, much lower, testifying that the distribution is very skewed. It should also be noted that the number of clients sending queries to these endpoints (column *Hosts* of the table) in that period is remarkably small. They are 2,306 in total, with many of the endpoints being used by less than one hundred clients. The global average number of query executions per host is $\sim$15K. This probably at least partially explains the high repetition of queries and the skewness of their distribution, as the usage is dominated by few clients which probably perform repetitive tasks.

1.1 Results

Regarding Bio2RDF, the templates generated (column *Templ.s*) are globally $\sim$12K, more than two orders (base 10) of magnitude smaller than the number of queries, which corresponds in base 2 to 7.27 orders of magnitude (column

[19] Most of the Bio2RDF logs cover the entire period of time, while some of them cover just a portion, possibly due to unavailability of those datasets

Dataset	**Execs**	**Hosts**	**Queries**	**H(Q)**	**Templ.s**	**H(T)**	**ΔH**
Bio2RDF	**33 829 184**	**2 306**	**1 899 027**	**15.22**	**12 296**	**3.73**	**11.49**
Affymetrix	1 232 713	400	311 096	10.46	777	2.39	8.07
BioModels	1 239 915	183	435 232	14.31	517	2.34	11.97
BioPortal	1 337 805	60	89 664	12.80	198	1.57	11.22
CTD	942 021	285	287 296	12.30	1 143	2.22	10.08
dbSNP	794 460	32	269 498	15.41	192	1.50	13.91
DrugBank	1 616 082	999	379 234	13.66	3 873	4.43	9.23
GenAge	589 410	34	265 067	16.10	215	1.14	14.96
GenDR	691 486	33	270 697	15.60	216	1.80	13.79
GO	1 842 035	237	121 542	13.74	2 360	2.34	11.41
GOA	3 548 166	217	343 836	14.36	612	2.49	11.87
HGNC	1 532 705	345	364 961	11.78	808	3.00	8.77
NCBI Homologene	1 246 306	897	321 061	10.54	836	2.41	8.13
iRefIndex	1 562 102	80	309 777	13.74	552	2.34	11.40
KEGG	66 832	205	19 871	13.11	474	3.86	9.25
LinkedSPL	337 001	19	204 112	16.99	117	0.21	16.78
MGI	1 319 576	270	319 627	10.97	702	2.47	8.51
NCBI Gene	770 716	38	216 832	15.79	375	1.27	14.51
OMIM	1 510 163	403	335 541	11.64	2 579	2.97	8.67
PharmGKB	94 542	63	24 000	12.51	154	3.34	9.17
SABIORK	925 409	51	274 098	11.26	313	1.65	9.61
SGD	974 412	309	318 641	15.01	972	2.93	12.08
SIDER	599 914	55	277 766	16.10	455	1.39	14.71
Taxonomy	7 701 880	89	354 582	11.37	409	1.01	10.35
WormBase	1 353 533	37	498 170	17.62	284	3.28	14.33
DBpedia	**6 999 815**	**37 056**	**4 257 903**	**21.16**	**17 715**	**5.58**	**15.59**
DBpedia-2010	518 717	1 649	358 955	17.99	2 223	5.66	12.33
DBpedia-2015/6	6 481 098	35 407	3 903 734	21.01	15 808	5.21	15.80
Wikidata	**3 298 254**	**-**	**844 260**	**12.26**	**167 578**	**7.47**	**4.80**
LinkedGeoData	**501 197**	**25 431**	**173 043**	**14.24**	**2 748**	**4.78**	**9.46**
SWDF	**1 415 568**	**921**	**101 422**	**14.54**	**1 826**	**1.03**	**13.51**

Table 2: Statistics on the dataset (extended version).

$log_2 \frac{|Q|}{|T|}$). The entropy calculated on the template distribution (column H(T)) is 3.73. The information gain measured as difference in entropy amounts thus to 11.49 bits (column ΔH). Looking at specific Bio2RDF endpoints, the entropy reduction is noticeable across all of them, ranging from 8.07 bits (for *Affymetrix*) to 16.78 bits (for *LinkedSPL*).

A qualitative analysis of the most frequently executed templates shows that few of them account for most of the executions. Table 3 shows the ten most executed templates. For shortness of presentation the templates have been represented through functional prototype-like notation. As an example and for its relevance, the template TRIPLES(*subject*) is shown in its entirety in Query B.5. From a semantic point of view, it is a fairly general CONSTRUCT query. Nev-

Template	Execs	Queries	EPs	Hosts
TRIPLES(*subject*)	9 229 696	500 455	22/24	3
OBJECTS(*subject*, *property*)	7 948 291	55 535	17/24	13
DESCRIBE(*resource*)	5 808 471	162	21/24	76
CEPROTEINS(*geneSymReg*)	1 229 140	27 199	GOA	7
GENESSUBJECTOF(*property, object*)	1 156 637	27 203	5/24	7
PROCESSES(*gene*)	1 121 007	4 454	GOA	7
DIRECTCOMMONSUPERCLASSES(*class1*, *class2*)	839 519	16 357	BioPortal	1
SUBJECTSPREDICATES(*object*)	663 193	14 450	9/24	8
INTERACTIONS(*protein*)	609 345	7 965	iRefIndex	7
DIRECTSUPERCLASSES(*class*)	358 907	130	BioPortal	1

Table 3: Bio2RDF: most frequently executed templates.

ertheless, its syntax is distinct enough to conjecture, considering also its prominence and uniqueness (no other similar possible CONSTRUCT queries are found with high numbers of executions), that corresponds to a single query-source. Furthermore, this template is generated non only for the whole Bio2RDF, but also for most of endpoints taken separately.

Template B.5:

```
CONSTRUCT
  {
    ?s ?p ?o .
  }
WHERE
  { ?s  ?p  ?o
    FILTER ( ?s = $_1 )
  }
```

The second most executed template, OBJECTS(*subject*, *property*), is fully shown in Query B.6. This template is a very general one as well, this time also syntactically. Moreover, considering each endpoint at a time, in most of the cases more specific templates are recognised rather than this one. So this template probably does not correspond to a single query-source, but it is rather a commonly used pattern. Similar considerations can be drawn about SUBJECTSPREDICATES(*object*), a similarly generic template (the eighth most executed) which in a way performs the matching the other way around. While OBJECTS(*subject*, *property*) has been executed on 17 endpoints ∼8 million times, SUBJECTSPREDICATES(*object*) has been executed only on nine endpoints and roughly an order of magnitude less often.

Template B.6:

```
SELECT  ?o
WHERE
  { $_1
              $_2  ?o
  }
```

The third most executed template is a simple DESCRIBE of a single resource. This is also a candidate for convergence of multiple sources. Further analysis shows that most of the executions (∼5.3K) belong to a specific value for the parameter, namely `<http://localhost/ping>`, so that peculiar DESCRIBE should be considered on its own and there are good chances that comes from a single query-source.

Among the other templates, three of them, ceProteins(*geneSymReg*), processes(*gene*), and interaction(*protein*) are highly specific templates found only in the logs of specific endpoints: the first one, found in GOA, look for proteins of a species of nematode (*Caenorhabditis elegans*), using a regex over the gene id (i.e. *geneSymReg*); the second one, also used querying GOA, lists cellular processes associated to a specific gene; the last one, used in iRefIndex, looks for the proteins interacting with a given one and the associated resources.

Another one, genesSubjOf(*property,object*), consists of a single triple pattern, as it was the case for objects(*subject*, *property*). This time the predicate and object are parametric and the subject is the variable. Interestingly, the name given to the variable used for the subject is less generic: `?gene`. Still, the template is probably a case of convergence given that the label *gene* is arguably common in this domain and that, as in the case of objects(*subject*, *property*), more specific templates are recognised in place of this one in the endpoints.

The templates directSuperClasses(*class*) and directCommonSuperClasses(*class1*,*class2*), finally, are found in BioPortal and list all the direct super classes of either just *class* or both *class1* and *class2*. While the intent of these templates is not specific to the domain or endpoint, the fact that they occur only in one endpoint suggest the queries come from a single query-source.

2 DBpedia

LSQ contains two DBpedia query logs: *(i)* a log (indicated as *2010*) of the queries continuously collected from the 30th April 2010 to the 20th July 2010; *(ii)* a log (indicated as *2015/6*) which is the result of the composition of 13 one-day logs collected between the 25th of October 2015 and the 11th of April 2016. The number of query executions (log entries) in the first dataset is ∼519K, while in the second one is ∼6.48M even if the latter covers a smaller amount of time (13 days instead of 81 days). This fact testifies a sharp increase in usage of DBpedia, from ∼6.4K query executions per day in 2010 to ∼500K in 2015/2016. The number of hosts increases significatively too, from ∼1.6K to ∼35K, showing much broader usage[20]. The number of unique queries in both datasets is comparable to the number of executions, respectively ∼359K and ∼3.9M, showing high variety. The query variety is confirmed by the entropy values, which are close to the maximum values, i.e. $log_2(|Q|)$, for both the datasets as well as for them as a whole.

[20] It should be noted that the hosts from 2010 and the ones from 2015/6 are anonymised in different ways, so there is no overlap between the two sets. The total number shown for hosts is thus simply the sum between the corresponding values of each dataset.

2.1 Results

The templates identified in the two logs are respectively ∼2.2K and ∼16K. The summarisation through templates is thus able to simplify the dataset by a significant order of magnitude (7.91 in base 2, considering the whole dataset). The gain in terms of entropy is higher (15.59) showing that the process is able to merge the most significant (by number of executions) queries. It should anyway be noted the entropy of the template distribution (5.58), while much lower than the one of the query distribution, is still higher than in the Bio2RDF case, showing more diversity, in accordance with the much higher number of hosts and broader range of usage of the DBpedia knowledge graph.

Furthermore, it is worth noticing that, while there is a significant difference in the query entropy from the data of 2010 (17.99) to the ones of 2015/6 (21.01), in line with a ten-fold increase in both executions and unique queries, the respective entropies measured on templates distribution are much closer, actually sightly decreasing from 2010 (5.66) to 2015/6 (5.21). This is interesting because it shows that the template diversity remains stable, while the number and diversity of specific queries increase roughly as the volume of the executions. In our opinion this case also manifests the importance of using the entropy as an index of diversity, rather than just counting the total number templates (which is instead quite different between the two datasets, ∼2.2K against ∼16K).

Template	**Execs**	**Queries**	**Hosts**	**Period**
airportsForCity(*cityLabel*, *lang.*)	1 384 760	378 390	12 012	all
describe(*resource*)	1 289 555	1 074 265	8 856	all
cityInfo(*cityLabel*, *language*)	624 194	79 878	7 534	2015/6
objects(*subject*, *predicate*)	376 786	363 682	5 480	all
countLinksBetween(*res1*, *res2*)	181 823	169 632	1 129	only 2015-12-29
countCommonLinks(*res1*, *res2*)	181 227	169 008	1 135	only 2015-12-29
objectsAsTypes(*subj*, *pred*)	115 918	102 612	1 529	all
personAndTypes(*wikiPageID*)	83 082	81 128	313	only 2016-02-11
labelAndLocation(*poi*)	81 285	63 559	432	all
searchByLabel(*string*)	70 109	68 345	341	only 2015-11-23

Table 4: DBpedia: most frequently executed templates.

Table 4 shows the signature of the ten most executed templates[21], considering the DBpedia logs as a whole. Two of them, airportsForCity(*cityLabel*, *language*) and cityInfo(*cityLabel*, *language*), are highly specific, returning respectively a list of airports and general information about a city with a given name. The parameter *language* is used to select the language used for the returned

[21] The names of the templates and their parameters were assigned by us, while the actual body of the templates can be found as supplemental material (see title note and note 16), namely in dbpedia.csv inside templatesAsCSV.zip

labels. Albeit being specific, they are both executed by a significant number of hosts (respectively ∼12K and ∼7.5K) and for a broad period.

The templates DESCRIBE(*resource*), OBJECTS(*subject*, *predicate*), and OBJECTSASTYPES(*subject*, *predicate*), on the other hand, are simple queries that can be conjectured to be widely used when querying SPARQL endpoints. The number of hosts and their recurrence over a large period of time testify indeed the broad use of these templates.

The templates COUNTLINKSBETWEEN(*res1*, *res2*) and COUNTCOMMONLINKS(*res1*, *res2*), finally, are non-trivial, albeit the purpose is generic. They both take two resources *res1* and *res2* as parameters and count respectively the triples between the two and the triples in which they replace each other or have a symmetric role. Given the complexity, they are probably single query-source templates. Moreover, the similar number of corresponding queries and executions hints at the fact that they are executed on an almost completely overlapping collection of pairs of resources. The fact that they are executed both only on a specific day (among the ones available in the dataset), further suggests a relationship between them.

The last three templates of this list, finally, are halfway in terms of specificity, since they solve common use cases but they are not as generic as the DESCRIBE() or OBJECTS(). The template PERSONANDTYPES(*wikiPageID*) return humans (and their associated types) corresponding to a certain Wikipedia page, while LABELANDLOCATION(*poi*) gathers the label and geographic coordinates of a resource, and SEARCHBYLABEL(*string*) looks for resources whose label contains a given string, using for that purpose a SPARQL syntax specific to OpenLink Virtuoso[22]. The template LABELANDLOCATION() is used in the whole available period, while the other two are just used in one specific day each.

3 Wikidata

The Wikidata data come from a filtered release of the log from the 11th of June 2017 to the 25th of March 2018. In the original release, the providers distinguish between queries probably issued by an automated process, called *robotic* traffic, and those which are not, called *organic* traffic. The LSQ dataset contains just the organic part. Identification of the clients is not available in this case. Furthermore, the Wikidata maintainers performed a process of anonymisation that includes rewriting the queries so that the variables have normalised names, rather than the ones originally used.

The dataset contains ∼3.3M query executions of 844K queries, putting its query diversity halfway between Bio2RDF and DBpedia. Nevertheless, the entropy value is comparatively low, similar to the values for Bio2RDF, showing that the query distribution is very skewed. It should be noted that the anonymi-

[22] As other triple stores, OpenLink Virtuoso defines a set of "magic properties", which can be used as a way to execute specific functions inside queries. In this case, the magic property `bif:contains` is used for full-text search.

sation process may have an impact, by unifying syntactically different queries through the normalisation of the variable names.

3.1 Results

Also for Wikidata logs, the representation with templates simplifies significatively the dataset. The templates are ∼168K for ∼844K unique queries and the difference in entropy amounts to 4.80 bits. The gain is lower than for the other datasets, in part because of the high repetition of queries in the log and consequent already low entropy of the query distribution, in part possibly because this is a pre-filtered log where only the *organic* queries have been picked.

Template	**Execs**	**Queries**
closestCityAndAirport(*location*)	776 647	202
searchHumans(*name*)	465 719	10 535
commonSuperClassAndDistance(*class1*, *class2*)	107 161	76 886
poisInArea(*corSW*, *corNE*, *langs*)	62 089	2 659
subjects(*property*, *object*)	53 546	17 979
distinctSubjects(*property*, *object*)	48 825	1 659
qualifiedStatements(*property*, *object*)	47 303	10 859
existsTripleWith(*property*, *object*)	46 557	27 103
objectsFromSubject(*property*, *object*, *otherProperty*)	46 193	21 287
subjectsObjects(*property*, *object*)	43 547	30 948

Table 5: Wikidata: most frequently executed templates.

Table 5 lists the ten most executed templates in the Wikidata log. Three of them, closestCityAndAirport(*location*), searchHumans(*name*), and poisInArea(*cornerSW*,*cornerNE*,*language*), are quite specific, respectively looking for the city (with an airport) closest to *location*, listing humans with a certain *name*, and returning all the points of interest (POI) in a specified area.

On the other hand, the template commonSuperClassAndDistance(*class1*,*class2*) is generic and similar in purpose to directCommonSuperClass() in Bio2RDF log. In this case, instead of just listing the direct super classes of both, the full subclass hierarchy going up is considered, picking the closest common super class and measuring the distance from one class to the other. Albeit it would be possible to write it in standard SPARQL, the query uses GAS API, an extension available on Blazegraph, the triple store used by Wikidata, probably in order to achieve greater efficiency.

The other six templates among the most executed are also functionally generic. Interestingly they are all constructed in a similar way: given a *property* and a *object* they identify each resource that is subject of a statement including *property* as predicate and *object* as object. While subjects and distinctSubjects directly return that set of resources, one with repetitions and the other without, the templates objectsFromSubject and subjectsObjects further explore the graph from that set of resources, in a case traversing *anotherProperty* in the other the same *property* already used to define the set. The template qualifiedStatements(*property*,*object*) is peculiar because, albeit content-wise generic, is specific to the Wikidata model as it follows the idiosyncratic way of performing the reification of statements in Wikidata: there are multiple IRIs associated with a single property, depending on the way it its used; in this template for each subject and property identified as in the previous cases, one of the available statement qualifiers ("properties" of the statement) is gathered. The simplest of this group of templates is existsTripleWith, which just checks for the existence of at least a triple with *property* as predicate and *object* as object. It is remarkable for being, among the most used templates of all the endpoints, the only one modelling ASK queries.

Albeit not among the most executed templates, a significant number of templates found in Wikidata (1049) have placeholders used in `VALUES` clauses. Template B.7 is the most executed among them, with 26443 executions and 21772 different instances. Given a set of Twitter accounts specified by the respective Twitter username, returns IRIs of people and organisations operating those (the Wikidata property `P2002` associates some resource with a corresponding Twitter username).

Template B.7:

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wikibase: <http://wikiba.se/ontology#>
SELECT  ?var1 ?var2
WHERE
  { ?var1  wdt:P2002           ?var2 ;
           wikibase:sitelinks  ?var3
    VALUES ?var2 $_1
  }
ORDER BY ASC(?var3)
```

4 LinkedGeoData

The LinkedGeoData log included in LSQ covers a year, precisely the period from the 22nd of November 2015 to the 20th of November 2016.

It contains ∼501K executions of ∼173K unique queries, a proportion similar to the Wikidata endpoint, but with a comparatively more uniform distribution. In respect to the other logs, the one from LinkedGeoData contains a small number of executions, but interestingly it was queried by a non-negligible number of hosts (∼25K).

4.1 Results

The summarisation is able to reduce the ∼173K queries to ∼2.7K templates, with a difference in entropy that amounts to 9.46.

Template	**Execs**	**Queries**	**Hosts**
DISTINCTOBJECTSOPT(*subject*, *predicate*)	110 650	4 367	2
CLOSEPOIS(*latitude*, *longitude*)	80 633	1 568	22776
PREDICATESOBJECTSOPT(*subject*)	40 243	22 432	1
SUBJECTSPREDICATESOBJECTS(*limit*, *offset*)	36 228	21 545	6
COMMONSUBCLASSES(*class1*, *class2*, *class3*)	34 412	34 412	1
COMMONSUBCLASSES(*class1*, *class2*)	16 627	13 944	2
ROISATPOSITION(*latitude*, *longitude*)	13 309	549	1
PREDICATESOBJECTS(*subject*)	11 078	5 701	22
ALLCLASSES(*offset*)	8 801	1 144	1
COUNTFEATURES(*property*)	8 786	8 786	1

Table 6: LinkedGeoData: most frequently executed templates.

The templates with most executions are shown in Table 6. They are all quite generic in purpose, but most of them are quite specific in the way they are written. Specifically, DISTINCTOBJECTSOPT(*subject*, *predicate*) and PREDICATESOBJECTSOPT(*subject*) are both composed of single triple patterns, but curiously embedded inside a redundant OPTIONAL clause[23]. The template PREDICATESOBJECTS(*subject*) is the non redundant version of PREDICATESOBJECTSOPT.

Perhaps unsurprisingly, given the content of LinkedGeoData, two of the most used templates perform geospatial queries: CLOSEPOIS(*latitude*,*longitude*) lists the POIs close to (strictly less than 2 km) the specified location, while ROISATPOSITION(*latitude*,*longitude*) lists the regions of interest which include the specified location. Both of them use OpenLink Virtuoso geospatial extensions.

Other two templates, of the form COMMONSUBCLASSES(*class1*, *class2*, ...), list all the classes for which it exists at least an instance that has also types *class1*, *class2*, ... For each such class the number of instances is given. These two templates differ just for a line (there is one more in the one with three parameters). Furthermore, among the templates, a similar one with just one parameter can be found. These three templates are thus probably specific cases of a more general single query-source template supporting a variable number of parameters.

The template SUBJECTSPREDICATESOBJECTS(*limit*, *offset*) is composed of the simplest triple pattern, with variables `?s`, `?p`, and `?o`, modified with parametric LIMIT and OFFSET clauses. Being a pretty common query (e.g., to start exploring an unknown dataset), there are no grounds, in this case, to consider this a single query-source template. A similarly simple pattern is used by ALLCLASSES(*offset*) to get the classes in the dataset. It is more specific as the LIMIT clause has a fixed value, ten thousand, while the offset varies.

[23] The pattern obtained by embedding another pattern in an OPTIONAL clause (without any preceding pattern outside) is basically equivalent to the original one. Technically, it differs just in the case in which the inner pattern does not match, which would lead to an empty solution sequence in the original one versus a solution composed by one empty mapping using the OPTIONAL.

Finally, countFeatures(*property*) was probably meant to count geographical features reachable through a *property.* The interesting fact about this template is that, even if it was executed almost nine thousands times, it certainly does not work as intended. The variable used in the COUNT is not defined in the WHERE clause, so the result is always zero.

5 Semantic Web Dog Food

SWDF data contain six months of log, from the 15th of May to the 12th of November 2014.

The number of clients querying the service in that period is quite small (921), which probably explains a proportion between query executions (∼1.42M) and unique queries (∼101K) similar to the Bio2RDF case.

5.1 Results

The Semantic Web Dog Food (SWDF) log is characterised by being used by a relatively small set of hosts. The inferred templates are ∼1.8K, from 101K queries. The variation in entropy in bits is 13.51. Even more than the difference, it is quite impressive that the entropy after summarisation amounts to just 1.03 bits.

Template	**Execs**	**Queries**	**Hosts**
describe(*resource*)	1 278 419	26 535	10
propertiesAndValues(*resource*)	17 283	17 283	2
authorsAbstractKeywords(*article*)	9 702	4 885	1
info(*person*)	6 988	6 267	1
affiliationsAtEvents(*employee*)	6 974	6 267	1
rolesAtEvents(*person*)	6 974	6 267	1
publicationsAtEvents(*person*)	6 974	6 267	1
authorList(*article*)	5 234	4 863	1
metadata(*conferenceArticle*)	5 233	4 862	1
subjectPredicatesObjects(*limit*)	4 999	13	18
metadata(*organization*)	3 162	2 866	1
membersAtEvents(*organization*)	3 161	2 866	1

Table 7: SWDF: most frequently executed templates.

The explanation of this extreme value can be found through qualitative analysis of the most executed templates (in Table 7 are listed the ones with more than 1K executions): the most executed template, a simple DESCRIBE of a resource, is responsible for ∼1.3K executions over ∼1.4K, ∼90.3 % of them. As this template, discussed also for Bio2RDF, is the simplest form of DESCRIBE could very well come from different sources. Still, the proportion of executions suggest there is something specific to SWDF, either for a particular usefulness of the DESCRIBE on this dataset or because of some piece of software generating a large number of DESCRIBE queries.

The distribution of the executions among the other templates is more uniform. The second most executed template is PROPERTIESANDVALUES(*resource*), a simple triple pattern analogous to templates found for other endpoints. Another generic template found is SUBJECTSPREDICATESOBJECTS(*limit*), similar to the one found in LinkedGeoData, but without the offset parameter. These are common queries and thus quite probably cases of convergence.

The other templates with a high number of executions are very specific and retrieve information typical of the SWDF dataset, as articles, researchers, events, and affiliations. By looking at the number of queries and executions of these templates is possible to see that are groups of templates that have equal or very close numbers. Furthermore, inside each of these groups the input type is the same: four return information about a researcher and are executed 6982 ± 6 times; two give information about an article and are executed 5233 ± 1 times; two look for information about an organisation and are executed 3161 ± 1 times[24]. This suggests they are executed on the same data and are probably part of the same process.

[24] While we in general discussed in each case the ten most used templates, in this case we discussed the top twelve to better explore these relationships.

Appendix C

Full Code of the Most Executed Templates

This appendix contains, for convenience, the full code of each of the templates that were discussed in the paper. Please note that, as previously mentioned, the full dataset of extracted templates is publicly available online in multiple formats.

It is also worth recalling that the function prototype style label adopted to identify each template (e.g., cityInfo(*cityLabel*, *language*)) has been given based on our interpretation of the template and it is not a product of the algorithm. The code of each template is instead shown exactly as elicited by the algorithm. That is why in the code the template parameters have generic labels (e.g., `$__PARAM_0`, `$__PARAM_1`, ...). The association with the names given to the parameters in the template label should be straightforward.

1 Bio2RDF

Template C.8: triples(*subject*)

```
PREFIX bio2rdf: <http://bio2rdf.org/>
CONSTRUCT
  {
    ?s ?p ?o .
  }
WHERE
  { ?s  ?p  ?o
    FILTER ( ?s = $__PARAM_0 )
  }
```

Template C.9: objects(*subject*, *property*)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bio2rdf: <http://bio2rdf.org/>
PREFIX hgnc_voc: <http://bio2rdf.org/hgnc_vocabulary:>
SELECT  ?o
WHERE
  { $__PARAM_0
              $__PARAM_1  ?o
  }
```

Template C.10: describe(*resource*)

```
DESCRIBE $__PARAM_0
```

Template C.11: ceProteins(*geneSymReg*)

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX bio2rdf: <http://bio2rdf.org/>
```

```
SELECT  *
WHERE
  { ?protein  bio2rdf:goa_vocabulary:gene_symbol  ?goasymbol ;
              bio2rdf:goa_vocabulary:taxid  bio2rdf:taxon:6239
    FILTER regex(?goasymbol, $__PARAM_0)
  }
```

Template C.12: genesSubjectOf(*property, object*)

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX bio2rdf: <http://bio2rdf.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX vocab: <http://bio2rdf.org/ctd_vocabulary:>
SELECT  *
WHERE
  { ?gene  $__PARAM_0  $__PARAM_1 }
```

Template C.13: processes(*gene*)

```
PREFIX bio2rdf: <http://bio2rdf.org/>
SELECT  *
WHERE
  { $__PARAM_0
              bio2rdf:goa_vocabulary:process  ?goTerm
  }
```

Template C.14: directCommonSuperClasses(*class1*,*class2*)

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX bio2rdf: <http://bio2rdf.org/>
SELECT  ?higher
WHERE
  { $__PARAM_0
              rdfs:subClassOf  ?higher .
    $__PARAM_1
              rdfs:subClassOf  ?higher
  }
```

Template C.15: subjectsPredicates(*object*)

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT  ?S ?P
WHERE
  { ?S  ?P  $__PARAM_0 }
```

Template C.16: interactions(*protein*)

```
PREFIX bio2rdf: <http://bio2rdf.org/>
SELECT  *
WHERE
  {   { ?interaction  bio2rdf:irefindex_vocabulary:interactor_a  $__PARAM_0
       ;
                  bio2rdf:irefindex_vocabulary:interactor_b  ?otherProtein ;
                  bio2rdf:irefindex_vocabulary:number-supporting-articles  ?
                       articles ;
                  bio2rdf:irefindex_vocabulary:method  ?method
      }
    UNION
      { ?interaction2
                  bio2rdf:irefindex_vocabulary:interactor_a  ?otherProtein ;
                  bio2rdf:irefindex_vocabulary:interactor_b  $__PARAM_0 ;
```

```
                bio2rdf:irefindex_vocabulary:number-supporting-articles  ?
                    articles ;
                bio2rdf:irefindex_vocabulary:method  ?method
    }
  }
```

Template C.17: directSuperClasses(*class*)

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX bio2rdf: <http://bio2rdf.org/>
SELECT  ?higher
WHERE
  { $__PARAM_0
              rdfs:subClassOf  ?higher
  }
```

2 DBpedia

Template C.18: airportsForCity(*cityLabel*, *lang*)

```
PREFIX dbpo: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbprop: <http://dbpedia.org/property/>
SELECT  *
WHERE
  { ?city     rdf:type    dbpo:Place ;
              rdfs:label  $__PARAM_0 .
    ?airport  rdf:type    dbpo:Airport
      { ?airport  dbpo:city  ?city }
    UNION
      { ?airport  dbpo:location  ?city }
    UNION
      { ?airport  dbprop:cityServed  ?city }
    UNION
      { ?airport  dbpo:city  ?city }
      { ?airport  dbprop:iata  ?iata }
    UNION
      { ?airport  dbpo:iataLocationIdentifier  ?iata }
    OPTIONAL
      { ?airport  foaf:homepage  ?airport_home }
    OPTIONAL
      { ?airport  rdfs:label  ?name }
    OPTIONAL
      { ?airport  dbprop:nativename  ?airport_name }
    FILTER ( ( ! bound(?name) ) || langMatches(lang(?name), $__PARAM_1) )
  }
```

Template C.19: describe(*resource*)

```
DESCRIBE $__PARAM_0
```

Template C.20: cityInfo(*cityLabel*, *language*)

```
PREFIX dbpo: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX gsp: <http://www.opengis.net/ont/geosparql#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbprop: <http://dbpedia.org/property/>
SELECT  *
WHERE
  {   { ?city  rdfs:label  $__PARAM_0 }
    UNION
      { ?alias  dbprop:redirect  ?city ;
                rdfs:label       $__PARAM_0
      }
    UNION
      { ?alias  dbprop:disambiguates  ?city ;
                rdfs:label            $__PARAM_0
      }
    OPTIONAL
      { ?city  dbpo:abstract  ?abstract }
    OPTIONAL
      { ?city  gsp:lat   ?latitude ;
               gsp:long  ?longitude
      }
    OPTIONAL
      { ?city  foaf:depiction  ?image }
    OPTIONAL
      { ?city  rdfs:label  ?name }
    OPTIONAL
      { ?city  foaf:homepage  ?home }
    OPTIONAL
      { ?city  dbpo:populationTotal  ?population }
    OPTIONAL
      { ?city  dbpo:thumbnail  ?thumbnail }
    FILTER langMatches(lang(?abstract), $__PARAM_1)
  }
```

Template C.21: OBJECTS(*subject*, *predicate*)

```
PREFIX dbpr: <http://dbpedia.org/resource/>
PREFIX dbpo: <http://dbpedia.org/ontology/>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbprop: <http://dbpedia.org/property/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT  *
WHERE
  { $__PARAM_0
              $__PARAM_1  ?o
  }
```

Template C.22: COUNTLINKSBETWEEN(*res1*, *res2*)

```
PREFIX dbpr: <http://dbpedia.org/resource/>
SELECT DISTINCT  (COUNT(?p) AS ?count)
WHERE
  {   { $__PARAM_0
                  ?p  $__PARAM_1
      }
    UNION
      { $__PARAM_1
                  ?p  $__PARAM_0
      }
  }
```

Template C.23: COUNTCOMMONLINKS(*res1*, *res2*)

```
PREFIX dbpr: <http://dbpedia.org/resource/>
SELECT DISTINCT  (COUNT(?p) AS ?count)
WHERE
  {   { ?o  ?p  $__PARAM_0 ;
            ?p  $__PARAM_1
      }
    UNION
      { ?o        ?p  $__PARAM_0 .
        $__PARAM_1
                  ?p  ?o
      }
    UNION
      { $__PARAM_0
                  ?p  ?o .
        ?o        ?p  $__PARAM_1
      }
    UNION
      { $__PARAM_0
                  ?p  ?o .
        $__PARAM_1
                  ?p  ?o
      }
  }
```

Template C.24: objectsAsTypes(*subj*, *pred*)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpr: <http://dbpedia.org/resource/>
PREFIX dbpo: <http://dbpedia.org/ontology/>
PREFIX planet: <http://dbpedia.org/>
PREFIX url: <http://schema.org/>
PREFIX dul: <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>
SELECT  ?type
WHERE
  { $__PARAM_0
              $__PARAM_1  ?type
  }
```

Template C.25: personAndTypes(*wikiPageID*)

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbpo: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT  ?person ?type
WHERE
  { ?person  dbpo:wikiPageID  $__PARAM_0 ;
             rdf:type         ?type
  }
```

Template C.26: labelAndLocation(*poi*)

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX dbpr: <http://dbpedia.org/resource/>
SELECT  ?label ?lat ?long
FROM <http://dbpedia.org>
WHERE
  { $__PARAM_0
              rdfs:label  ?label
    OPTIONAL
      { $__PARAM_0
                  pos:lat   ?lat ;
                  pos:long  ?long
      }
  }
```

Template C.27: SEARCHBYLABEL(*string*)

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT  ?x
WHERE
  { ?x  rdfs:label  ?name
    FILTER <bif:contains>(?name, $__PARAM_0)
  }
```

3 Wikidata

Template C.28: CLOSESTCITYANDAIRPORT(*location*)

```
PREFIX bd: <http://www.bigdata.com/rdf#>
PREFIX gsp: <http://www.opengis.net/ont/geosparql#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wikibase: <http://wikiba.se/ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX wde: <http://www.wikidata.org/entity/>
SELECT DISTINCT  ?var1 ?var1Label ?var2 ?var2Label ?var3 ?var4
WHERE
  { ?var2 wdt:P31/(wdt:P279)* wde:Q515 .
    ?var1  wdt:P31   wde:Q644371 ;
           ?var5     ?var2 ;
           wdt:P238  ?var3
    SERVICE wikibase:around
      { ?var2     wdt:P625           ?var6 .
        bd:serviceParam
                  wikibase:center    $__PARAM_0 ;
                  wikibase:radius    ""200"" ;
                  wikibase:distance  ?var7
      }
    SERVICE wikibase:label
      { bd:serviceParam
                  wikibase:language  ""en""
      }
    OPTIONAL
      { ?var2  wdt:P625  ?var4 }
  }
ORDER BY ASC(?var7)
LIMIT   1
```

Template C.29: SEARCHHUMANS(*name*)

```
PREFIX wde: <http://www.wikidata.org/entity/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT DISTINCT  ?var1
WHERE
  { ?var1  rdfs:label  $__PARAM_0 ;
           wdt:P31     wde:Q5
  }
```

Template C.30: COMMONSUPERCLASSANDDISTANCE(*class1*,*class2*)

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX wde: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT  ?var1 (( ?var2 + ?var3 ) AS ?var4)
```

```
WHERE
  { SERVICE <http://www.bigdata.com/rdf/gas#service>
      { <http://www.bigdata.com/rdf/gas#program>
                  <http://www.bigdata.com/rdf/gas#gasClass>  ""com.bigdata.
                       rdf.graph.analytics.SSSP"" ;
                  <http://www.bigdata.com/rdf/gas#in>  $__PARAM_0 ;
                  <http://www.bigdata.com/rdf/gas#traversalDirection>  ""
                       Forward"" ;
                  <http://www.bigdata.com/rdf/gas#out>  ?var1 ;
                  <http://www.bigdata.com/rdf/gas#out1>  ?var2 ;
                  <http://www.bigdata.com/rdf/gas#maxIterations>  10 ;
                  <http://www.bigdata.com/rdf/gas#linkType>  wdt:P279
      }
    SERVICE <http://www.bigdata.com/rdf/gas#service>
      { <http://www.bigdata.com/rdf/gas#program>
                  <http://www.bigdata.com/rdf/gas#gasClass>  ""com.bigdata.
                       rdf.graph.analytics.SSSP"" ;
                  <http://www.bigdata.com/rdf/gas#in>  $__PARAM_1 ;
                  <http://www.bigdata.com/rdf/gas#traversalDirection>  ""
                       Forward"" ;
                  <http://www.bigdata.com/rdf/gas#out>  ?var1 ;
                  <http://www.bigdata.com/rdf/gas#out1>  ?var3 ;
                  <http://www.bigdata.com/rdf/gas#maxIterations>  10 ;
                  <http://www.bigdata.com/rdf/gas#linkType>  wdt:P279
      }
  }
ORDER BY ASC(?var4)
LIMIT   1
```

Template C.31: poisInArea(*corSW*,*corNE*,*langs*)

```
PREFIX bd: <http://www.bigdata.com/rdf#>
PREFIX gsp: <http://www.opengis.net/ont/geosparql#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wikibase: <http://wikiba.se/ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX url: <http://schema.org/>
SELECT  ?var1 ?var1Label ?var2 ?var3 ?var4 ?var5 ?var6 ?var7
WHERE
  { SERVICE wikibase:box
      { ?var1     wdt:P625              ?var2 .
        bd:serviceParam
                  wikibase:cornerSouthWest  $__PARAM_0 ;
                  wikibase:cornerNorthEast  $__PARAM_1
      }
    OPTIONAL
      { ?var1  wdt:P18  ?var3 }
    OPTIONAL
      { ?var1  wdt:P373  ?var6 }
    OPTIONAL
      { ?var1  wdt:P969  ?var7 }
    SERVICE wikibase:label
      { bd:serviceParam
                  wikibase:language  $__PARAM_2 .
        ?var1     url:description    ?var5 ;
                  rdfs:label         ?var1Label
      }
  }
LIMIT   3000
```

Template C.32: subjects(*property*,*object*)

```
PREFIX wde: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX orth: <http://purl.org/net/orth#>
PREFIX wikibase: <http://wikiba.se/ontology#>
PREFIX url: <http://schema.org/>
PREFIX umbel: <http://umbel.org/umbel#>
PREFIX openlinks: <http://www.openlinksw.com/schemas/virtrdf#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX vcard2006: <http://www.w3.org/2006/vcard/ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX dbpr: <http://dbpedia.org/resource/>
PREFIX dbpo: <http://dbpedia.org/ontology/>
PREFIX dbprop: <http://dbpedia.org/property/>
PREFIX dby: <http://dbpedia.org/class/yago/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX wdv: <http://www.wikidata.org/value/>
PREFIX gsp: <http://www.opengis.net/ont/geosparql#>
PREFIX freebase: <http://rdf.freebase.com/ns/>
SELECT  ?var1
WHERE
  { ?var1  $__PARAM_0  $__PARAM_1 }
```

Template C.33: DISTINCTSUBJECTS(*property*,*object*)

```
PREFIX wde: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wikibase: <http://wikiba.se/ontology#>
PREFIX freebase: <http://rdf.freebase.com/ns/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX dcat: <http://www.w3.org/ns/dcat#>
PREFIX url: <http://schema.org/>
PREFIX wp: <http://vocabularies.wikipathways.org/wp#>
PREFIX dbpo: <http://dbpedia.org/ontology/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT  ?var1
WHERE
  { ?var1  $__PARAM_0  $__PARAM_1 }
```

Template C.34: QUALIFIEDSTATEMENTS(*property*,*object*)

```
PREFIX wde: <http://www.wikidata.org/entity/>
PREFIX wikibase: <http://wikiba.se/ontology#>
SELECT  ?var1 ?var2 (SAMPLE(?var3) AS ?var4)
WHERE
  { { SELECT DISTINCT  ?var1 ?var2
      WHERE
        { ?var2  $__PARAM_0  $__PARAM_1 .
          ?var1  $__PARAM_2  ?var2
        }
      LIMIT   101
    }
    OPTIONAL
      { ?var2  ?var3               ?var5 .
        ?var6  wikibase:qualifier  ?var3
      }
  }
GROUP BY ?var1 ?var2
```

Template C.35: EXISTSTRIPLEWITH(*property*,*object*)

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wde: <http://www.wikidata.org/entity/>
PREFIX space: <http://purl.org/net/schemas/space/>
ASK
WHERE
  { ?var1  $__PARAM_0  $__PARAM_1 }
```

Template C.36: objectsFromSubject(*property*,*object*,*otherProperty*)

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wde: <http://www.wikidata.org/entity/>
SELECT  ?var1
WHERE
  { ?var2  $__PARAM_0  $__PARAM_1
    OPTIONAL
      { ?var2  $__PARAM_2  ?var1 }
  }
```

Template C.37: subjectsObjects(*property*,*object*)

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX bd: <http://www.bigdata.com/rdf#>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX url: <http://schema.org/>
PREFIX wikibase: <http://wikiba.se/ontology#>
SELECT  ?var1 ?var2 ?var3
WHERE
  { { ?var1  $__PARAM_0  $__PARAM_1 }
    ?var2  url:about  ?var1
      { ?var2  url:inLanguage  ""en"" }
    UNION
      { ?var2  url:inLanguage  ""de"" }
    ?var1  $__PARAM_0  ?var3
    SERVICE wikibase:label
      { bd:serviceParam
                  wikibase:language  ""en""
      }
  }
```

4 LinkedGeoData

Template C.38: distinctObjectsOpt(*subject*, *predicate*)

```
PREFIX lgd: <http://linkedgeodata.org/triplify/>
PREFIX lgv: <http://linkedgeodata.org/ontology/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX ngeo: <http://geovocab.org/geometry#>
PREFIX pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT  ?v0
WHERE
  { OPTIONAL
      { $__PARAM_0
                  $__PARAM_1  ?v0
      }
  }
OFFSET  0
LIMIT   1000
```

Template C.39: closePois(*latitude*,*longitude*)

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT  ?uri ?label
WHERE
  { ?uri <http://geovocab.org/geometry#geometry>/<http://www.opengis.net/ont
      /geosparql#asWKT> ?pgv .
    ?uri  rdfs:label  ?label
    FILTER ( <bif:st_distance>(?pgv, <bif:st_point>($__PARAM_0, $__PARAM_1))
         < 2 )
  }
LIMIT   100
```

Template C.40: predicatesObjectsOpt(*subject*)

```
PREFIX lgd: <http://linkedgeodata.org/triplify/>
SELECT  ?p ?o
WHERE
  { OPTIONAL
      { $__PARAM_0
                  ?p  ?o
      }
  }
```

Template C.41: subjectsPredicatesObjects(*limit*, *offset*)

```

SELECT  *
WHERE
  { ?s  ?p  ?o }
OFFSET  $__PARAM_0
LIMIT   $__PARAM_1
```

Template C.42: commonSubclassesAndInstanceCount(*class1*, *class2*, *class3*)

```
PREFIX lgv: <http://linkedgeodata.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lgdm: <http://linkedgeodata.org/meta/>
PREFIX spatial: <http://geovocab.org/spatial#>
SELECT  ?c (COUNT(DISTINCT ?s) AS ?count)
WHERE
  { ?s  rdf:type  $__PARAM_0 ;
        rdf:type  $__PARAM_1 ;
        rdf:type  $__PARAM_2 ;
        rdf:type  ?c
  }
GROUP BY ?c
```

Template C.43: commonSubclassesAndInstanceCount(*class1*, *class2*)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lgv: <http://linkedgeodata.org/ontology/>
PREFIX spatial: <http://geovocab.org/spatial#>
PREFIX lgdm: <http://linkedgeodata.org/meta/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
```

```
SELECT  ?c (COUNT(DISTINCT ?s) AS ?count)
WHERE
  { ?s  rdf:type  $__PARAM_0 ;
        rdf:type  $__PARAM_1 ;
        rdf:type  ?c
  }
GROUP BY ?c
```

Template C.44: roisAtPosition(*latitude*, *longitude*)

```
PREFIX ngeo: <http://geovocab.org/geometry#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX gsp: <http://www.opengis.net/ont/geosparql#>
SELECT DISTINCT  ?class ?label ?s
WHERE
  { ?s     rdf:type       ?class ;
           rdfs:label     ?label ;
           ngeo:geometry  ?geom .
    ?geom  gsp:asWKT      ?g
    FILTER <bif:st_intersects>(?g, <bif:st_point>($__PARAM_0, $__PARAM_1),
         1.0E-6)
    FILTER NOT EXISTS { ?x  rdfs:subClassOf  ?class
                        FILTER ( ?x != ?class )
                      }
  }
```

Template C.45: predicatesObjects(*subject*)

```
PREFIX lgd: <http://linkedgeodata.org/triplify/>
SELECT  ?p ?o
WHERE
  { $__PARAM_0
              ?p  ?o
  }
```

Template C.46: allClasses(*offset*)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT  ?Concept
WHERE
  { ?s  rdf:type  ?Concept }
OFFSET  $__PARAM_0
LIMIT   10000
```

Template C.47: countFeatures(*property*)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lgv: <http://linkedgeodata.org/ontology/>
PREFIX openlinks: <http://www.openlinksw.com/schemas/virtrdf#>
PREFIX sd: <http://www.w3.org/ns/sparql-service-description#>
SELECT  (COUNT(?p) AS ?no)
WHERE
  { ?s  $__PARAM_0  ?o .
    ?o  rdf:type              lgv:Feature
  }
GROUP BY ?p
```

5 Semantic Web Dog Food

Template C.48: DESCRIBE(*resource*)

```
DESCRIBE $__PARAM_0
```

Template C.49: PROPERTIESANDVALUES(*resource*)

```
PREFIX swperson: <http://data.semanticweb.org/person/>
PREFIX dbpr: <http://dbpedia.org/resource/>
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>
PREFIX wn: <http://xmlns.com/wordnet/1.6/>
PREFIX rdfdf: <http://www.openlinksw.com/virtrdf-data-formats#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX openlinks: <http://www.openlinksw.com/schemas/virtrdf#>
PREFIX ical: <http://www.w3.org/2002/12/cal/ical#>
PREFIX planet: <http://dbpedia.org/>
SELECT  ?property ?value
WHERE
  { $__PARAM_0
              ?property  ?value
  }
```

Template C.50: AUTHORSABSTRACTKEYWORDS(*article*)

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>
PREFIX dce: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT  ?abstract ?keyword ?author_name
WHERE
  {   { $__PARAM_0
                  swrc:author  ?author
      }
    UNION
      { $__PARAM_0
                  foaf:maker  ?author
      }
    ?author  foaf:name  ?author_name
    OPTIONAL
      { $__PARAM_0
                  swrc:abstract  ?abstract
      }
    OPTIONAL
      {   { $__PARAM_0
                      swrc:keywords  ?keyword
          }
        UNION
          { $__PARAM_0
                      dce:subject  ?keyword
          }
        UNION
          { $__PARAM_0
                      swc:hasTopic  ?topic .
            ?topic    rdfs:label    ?keyword
          }
      }
  }
```

Template C.51: INFO(*person*)

```
PREFIX swperson: <http://data.semanticweb.org/person/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT  ?name ?homepage ?mbox_sha1sum ?page ?sameAs ?seeAlso
WHERE
  { $__PARAM_0
              rdf:type  foaf:Person
      { $__PARAM_0
                  foaf:name  ?name
      }
    UNION
      { $__PARAM_0
                  rdfs:label  ?name
      }
    OPTIONAL
      { $__PARAM_0
                  foaf:mbox_sha1sum  ?mbox_sha1sum
      }
    OPTIONAL
      { $__PARAM_0
                  foaf:homepage  ?homepage
      }
    OPTIONAL
      { $__PARAM_0
                  foaf:page  ?page
      }
    OPTIONAL
      { $__PARAM_0
                  owl:sameAs  ?sameAs
      }
    OPTIONAL
      { $__PARAM_0
                  rdfs:seeAlso  ?seeAlso
      }
  }
```

Template C.52: AFFILIATIONSATEVENTS(*employee*)

```
PREFIX swperson: <http://data.semanticweb.org/person/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>
SELECT DISTINCT  ?affiliation_url ?affiliation_name ?event_uri ?
    event_acronym ?prefLabel
WHERE
  { GRAPH ?g
      { $__PARAM_0
                  swrc:affiliation  ?affiliation_url
      }
    ?affiliation_url
              foaf:name          ?affiliation_name .
    ?event_uri  swc:completeGraph  ?g ;
              swc:hasAcronym     ?event_acronym
    OPTIONAL
      { ?affiliation_url
                  skos:prefLabel  ?prefLabel
      }
  }
ORDER BY ?event_acronym
```

Template C.53: ROLESATEVENTS(*person*)

```
PREFIX swperson: <http://data.semanticweb.org/person/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>
SELECT DISTINCT  ?event_uri ?event_acronym ?role_uri ?role_label
WHERE
  { GRAPH ?g
      {   { $__PARAM_0
                      swc:holdsRole  ?role_uri
          }
        UNION
          { ?role_uri  swc:heldBy  $__PARAM_0 }
        ?role_uri  rdfs:label  ?role_label
      }
    ?event_uri  swc:completeGraph  ?g ;
              swc:hasAcronym     ?event_acronym
  }
ORDER BY ?event_acronym
```

Template C.54: PUBLICATIONSATEVENTS(*person*)

```
PREFIX swperson: <http://data.semanticweb.org/person/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>
SELECT DISTINCT  ?publication_url ?publication_name ?event_uri ?
    event_acronym
WHERE
  { GRAPH ?g
      {   { $__PARAM_0
                      foaf:made  ?publication_url
          }
        UNION
          { ?publication_url
                      foaf:maker  $__PARAM_0
          }
        UNION
          { ?publication_url
                      dct:creator  $__PARAM_0
          }
          { ?publication_url
                      dct:title  ?publication_name
          }
        UNION
          { ?publication_url
                      rdfs:label  ?publication_name
          }
      }
    ?event_uri  swc:completeGraph  ?g ;
              swc:hasAcronym     ?event_acronym
  }
ORDER BY ?event_acronym
```

Template C.55: AUTHORLIST(*article*)

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX bibo: <http://purl.org/ontology/bibo/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrcext: <http://www.cs.vu.nl/~mcaklein/onto/swrc_ext/2005/05#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT  ?pred ?author_url ?author_name ?author_pref_label
WHERE
  { GRAPH ?graph
      {   { $__PARAM_0
                      bibo:authorList  ?authorList
          }
        UNION
          { $__PARAM_0
```

```
                      swrcext:authorList  ?authorList
          }
        ?authorList  ?pred  ?author_url
          { ?author_url  foaf:name  ?author_name }
        UNION
          { ?author_url  rdfs:label  ?author_name }
        OPTIONAL
          { ?author_url  skos:prefLabel  ?author_pref_label }
      }
  }
ORDER BY ?pred
```

Template C.56: METADATA(*conferenceArticle*)

```
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX dce: <http://purl.org/dc/elements/1.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT  ?name ?abstract ?webpage ?sameAs ?seeAlso ?event ?
    conference_uri ?conference_name ?conference_acronym ?keyword
WHERE
  { GRAPH ?graph
      {   { $__PARAM_0
                      dce:title  ?name
          }
        UNION
          { $__PARAM_0
                      dct:title  ?name
          }
        UNION
          { $__PARAM_0
                      rdfs:label  ?name
          }
        OPTIONAL
          { $__PARAM_0
                      swrc:abstract  ?abstract
          }
        OPTIONAL
          { $__PARAM_0
                      swrc:url  ?webpage
          }
        OPTIONAL
          { $__PARAM_0
                      rdfs:seeAlso  ?seeAlso
          }
        OPTIONAL
          { $__PARAM_0
                      owl:sameAs  ?sameAs
          }
        OPTIONAL
          { $__PARAM_0
                      swc:relatedToEvent  ?event
          }
      }
    OPTIONAL
      {   { $__PARAM_0
                      swrc:keywords  ?keyword
          }
        UNION
          { $__PARAM_0
                      dce:subject  ?keyword
          }
        UNION
          {   { $__PARAM_0
```

```
                          swc:hasTopic  ?topic
              }
            UNION
              { $__PARAM_0
                          foaf:topic  ?topic
              }
            ?topic  rdfs:label  ?keyword
          }
      }
    ?conference_uri
              swc:completeGraph  ?graph ;
              rdfs:label         ?conference_name ;
              swc:hasAcronym     ?conference_acronym
  }
```

Template C.57: SUBJECTPREDICATESOBJECTS(*limit*)

```

SELECT  *
WHERE
  { ?s  ?p  ?o }
LIMIT   $__PARAM_0
```

Template C.58: METADATA(*organization*)

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX swperson: <http://data.semanticweb.org/person/>
SELECT DISTINCT  ?name ?homepage ?page ?sameAs ?seeAlso ?latitude ?longitude
WHERE
  {   { $__PARAM_0
                  foaf:name  ?name
      }
    UNION
      { $__PARAM_0
                  rdfs:label  ?name
      }
    OPTIONAL
      { $__PARAM_0
                  foaf:page  ?page
      }
    OPTIONAL
      { $__PARAM_0
                  owl:sameAs  ?sameAs
      }
    OPTIONAL
      { $__PARAM_0
                  rdfs:seeAlso  ?seeAlso
      }
    OPTIONAL
      { $__PARAM_0
                  foaf:homepage  ?homepage
      }
    OPTIONAL
      { $__PARAM_0
                  foaf:based_near  ?location .
        ?location  pos:lat         ?latitude ;
                  pos:long         ?longitude
      }
  }
```

Template C.59: MEMBERSATEVENTS(*organization*)

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>
PREFIX swperson: <http://data.semanticweb.org/person/>
SELECT DISTINCT  ?member_url ?member_name ?event_uri ?event_acronym ?
    prefLabel
WHERE
  { GRAPH ?g
      {   { $__PARAM_0
                      foaf:member  ?member_url
          }
        UNION
          { ?member_url  swrc:affiliation  $__PARAM_0 }
      }
      { ?member_url  foaf:name  ?member_name }
    UNION
      { ?member_url  rdfs:label  ?member_name }
    ?event_uri  swc:completeGraph  ?g ;
              swc:hasAcronym     ?event_acronym
    OPTIONAL
      { ?member_url  skos:prefLabel  ?prefLabel }
  }
ORDER BY ?member_url ?event_acronym
```