SPORTAL: Profiling the Content of Public SPARQL Endpoints

Ali Hasnain[†] Qaiser Mehmood[†] Syeda Sana e Zainab[†] Aidan Hogan[‡]

[†]*INSIGHT Centre for Data Analytics, National University of Ireland, Galway*

[‡]*Center for Semantic Web Research, Department of Computer Science, University of Chile*

Abstract

Access to hundreds of knowledge-bases has been made available on the Web through public SPARQL endpoints. Unfortunately, few endpoints publish descriptions of their content (e.g., using VoID). It is thus unclear how agents can learn about the content of a given SPARQL endpoint or, relatedly, find SPARQL endpoints with content relevant to their needs. In this paper, we investigate the feasibility of a system that gathers information about public SPARQL endpoints by querying them directly about their own content. With the advent of SPARQL 1.1 and features such as aggregates, it is now possible to specify queries whose results would form a detailed profile of the content of the endpoint, comparable with a large subset of VoID. In theory it would thus be feasible to build a rich centralised catalogue describing the content indexed by individual endpoints by issuing them SPARQL (1.1) queries; this catalogue could then be searched and queried by agents looking for endpoints with content they are interested in. In practice, however, the coverage of the catalogue is bounded by the limitations of public endpoints themselves: some may not support SPARQL 1.1, some may return partial responses, some may throw exceptions for expensive aggregate queries, etc. Our goal in this paper is thus twofold: (i) using VoID as a bar, to empirically investigate the extent to which public endpoints can describe their own content, and (ii) to build and analyse the capabilities of a best-effort online catalogue of current endpoints based on the (partial) results collected.

1 Introduction

Linked Data aims at making data available on the Web in an interoperable format so that agents can discover, access, combine and consume content from different sources with higher levels of automation than would otherwise be possible [22]. The envisaged result is a "Web of Data": a Web of structured data with rich semantic links where agents can query in a unified manner – across sources – using standard languages and protocols. Over the past few years, hundreds of knowledge-bases with billions of facts have been published according to the Semantic Web standards (using RDF as a data model and RDFS and OWL to provide explicit semantics) following the Linked Data principles.

As a convenience for consumer agents, Linked Data publishers often provide a SPARQL endpoint for querying their local content [25]. SPARQL is a declarative query language for RDF in which graph pattern matching, disjunctive unions, optional clauses, dataset construction, solution modifiers, etc., can be used to query RDF knowledge-bases; the recent SPARQL 1.1 release adds features such as aggregates, property paths, sub-queries, federation, and so on [19]. Hundreds of public endpoints have been published in the past few years for knowledge-bases of various sizes and topics [25, 10]. Using these endpoints, clients can receive direct answers to complex queries using a single request to the server.

However, it is still unclear how clients (be they human users or software agents) should find endpoints relevant for their needs in the first place [36, 10]. A client may have a variety of needs when looking for an endpoint, where they may, for example, seek endpoints with data:

1. about a given resource, e.g., MICHAEL JACKSON;
2. about instances of a particular type of class, e.g., PROTEINS;
3. about a certain type of relationship between resources, e.g., DIRECTS-MOVIE;

4. about certain types of values associated with resources, e.g., RATING;
5. about resources within a given context or with specific values, for example, CRIMES WITH LOCATION U.K. IN YEAR 1967 or RAT GENES AND DISEASE STRAINS;
6. a combination of one or more of the above.

Likewise a client may vary in how they are best able to specify these needs: some clients may only have keywords; others may know the specific IRI(s) of the resource, class or property they are interested in; some may be able to specify concrete queries or sub-queries that they wish to answer.

We argue that a service offering agents the ability to find relevant public endpoints on the Web would serve as an important part of the SPARQL querying infrastructure, enabling ad-hoc discovery of datasets over the Web. However, realising such a service over the current SPARQL infrastructure on the Web is challenging. Looking at the literature (in particular, works on the related problem of federated querying [43, 20, 2, 39]), we can find two high-level approaches that have been investigated thus far:

Runtime queries: The first option is to take an agent's request and query the endpoints directly at runtime to determine if they have relevant metadata or not [43]. For example, if the agent were interested in instances of `mo:MusicalWork`,[1] one could issue a list of endpoints the following query:

```
ASK WHERE { ?s a mo:MusicalWork }
```

Any endpoint returning `true` for this query would contain information relevant to the original agent. Likewise, more complex queries could be used depending on the user's need. For example, if a user were interested in endpoints with more than 100 such instances, the service could issue:

```
SELECT (COUNT(DISTINCT ?s) AS ?c) WHERE { ?s a mo:MusicalWork }
```

Any endpoint returning a result greater than 100 would be relevant.

Published content descriptions: The second option is to rely on a static description of the content of each endpoint [43, 20, 2, 39]. These works either assume that a description is available in a popular format, such as the Vocabulary of Interlinked Datasets (VoID [4]), or a custom format [20, 2, 39]. For example, the VoID vocabulary allows for defining class partitions that not only state which classes are in a dataset, but how many instances it has, which properties appear, and so forth [4]. These descriptions can then be used directly to find endpoints with relevant content.

However, these approaches are themselves problematic.

With respect to the first approach, each user request would require a query to be sent to several hundred public endpoints, which would incur very slow response times and could flood public services with too many requests. Likewise, users would need to know the IRIs of the resources, classes and/or properties they are interested in where supporting keyword search would be cumbersome to support: (i) although SPARQL does support functions such as `REGEX` that could be used to find relevant terms in literals, these functions are often executed as post-filtering operations, incurring unpredictable performance; (ii) although many SPARQL engines do build and maintain inverted indexes for efficient full-text search with keywords, this support is non-standard, different engines support full-text search in different manners, and determining the engine powering a SPARQL endpoint is non-trivial [10].

With respect to the second approach, Buil-Aranda et al. [10] previously observed that only one third of public SPARQL endpoints give static descriptions of their content in a standard location using suitable vocabularies such as VoID, and even where they are provided, it is unclear what level of detail these descriptions contain or indeed how accurate or up-to-date these descriptions are. Over the past several years, at least 159 distinct websites have begun hosting SPARQL endpoints [10]. Putting the burden on publishers to provide static descriptions of their endpoints' content or to otherwise change how they host data would incur a prohibitive technical and social cost.

For these reasons, in this paper we propose and explore the feasibility of a third approach:

[1]Note that all prefixes used in this paper are listed in Table 12 of the Appendix.

Computing content descriptions: Rather than relying on publishers to compute and keep content descriptions up to date, we propose to compute such descriptions directly from the endpoints themselves. In particular, we propose to design a set of queries that can be issued to endpoints to learn about their content, where the results of these queries can then be used to build a catalogue that enables clients to find endpoints with relevant content.

This approach offers a number of useful trade-offs when compared with the previous two approaches discussed earlier.

Comparing the use of computed content descriptions with runtime queries, in the former case, the client will query a centrally indexed catalogue, which incurs a lower cost, both for the client in terms of response time, and also for the remote SPARQL infrastructure in terms of the number of requests generated. However, the client will be restricted to finding endpoints using the metadata collected in the catalogue. For this reason, it is important for the catalogue to capture general descriptions of content.

When compared with using published content descriptions, we do not need to assume that such descriptions are provided by the publishers of SPARQL endpoints independently of the endpoint itself. Also, by computing the content descriptions, we ensure that the endpoint is still available (since it needs to answer the queries we send it), that the description is at least as recent as the last time the descriptions were computed, and that the statistics have a simple SPARQL query that acts as provenance (rather than using descriptions provided by the publishers themselves that could be produced by tools with, e.g., different interpretations of statistics or that may include manual approximations). However, it is not clear if public endpoints would be able to support the type of complex SPARQL query required to compute detailed content descriptions, and indeed certain types of descriptors (e.g., licence) may not be automatically computed from the endpoint but rather require the perspective of the publisher.

In this paper, we explore the feasibility of computing content descriptions directly from SPARQL endpoints. More concretely, we propose Sportal (SPARQL portal): a centralised catalogue indexing content descriptions computed from individual SPARQL endpoints. The goal of Sportal is to help both human and software agents find public SPARQL endpoints relevant for their needs. The system makes minimal assumptions about how data are hosted: Sportal relies only on SPARQL queries to gather information about the content of each endpoint and hence only assumes a working SPARQL interface rather than requiring the publishers hosting endpoints to provide additional descriptions of the datasets. Rather than send a query to each public endpoint at runtime, we issue each endpoint queries offline to gather metadata about its content, which are later used to find relevant endpoints. Taking a simple example, instead of querying each endpoint every time an agent is looking for a given class, we can occasionally query each endpoint (on a fortnightly basis) for an up-to-date list of their classes and use that list to find relevant endpoints for the agent at runtime.

One of the main design questions for Sportal then is: what content descriptions should such a system try to compute from endpoints? Ideally the content descriptions should be as general as possible, supporting a variety of different types of clients and searches. With respect to the information collected, SPARQL is a powerful query language that can be used to learn about the underlying knowledge-base of the endpoint. With the advent of novel features in SPARQL 1.1 like aggregates, it is now possible to formulate queries that ask, e.g., how many triples the knowledge-base contains, which classes or properties are used, how many unique instances of each class appears, which properties are used most frequently with instances of which classes, and so on. In this sense, we argue that – at least in theory – SPARQL endpoints can be considered *self-descriptive*: they can describe their own content.

On the other hand, Sportal is limited in what it can collect by practical thresholds on the amount of data that a SPARQL endpoint will return. Buil-Aranda et al. [10] found that many endpoints return a maximum of 10,000 results: given that many endpoints contain millions of resources and text literals, this rules out, for example, building a complete inverted index over the content of an individual endpoint, or indexing all resources that an endpoint mentions. In any case, the goal of Sportal is to compute concise content descriptions rather than mirroring remote endpoint content (which would be prohibitively costly for both Sportal and the remote endpoints, particularly to keep up-to-date). Thus, we focus on computing concise, schema-level descriptions of endpoints. Using such descriptions, we can directly find relevant endpoints given queries of type 2, 3, 4 mentioned earlier, and can indirectly help with other forms of queries (e.g., to find endpoints that contain instances of gene, though they may not necessary be from a rat). In particular, we focus on computing extended Vocabulary of Interlinked Datasets (VoID) descriptions from endpoints: VoID has become the de-facto standard for describing datasets in RDF [4], and is also used in federated scenarios to find relevant endpoints [39, 43, 3, 2, 6].

Sportal is further limited by the inability of some endpoints to return answers to complex queries. Buil-Aranda et al. [10] previously reported that endpoints may exhibit performance and reliability issues, may return partial results, etc. Some endpoints may not support SPARQL 1.1, some may be hosted on underpowered machines, others may index very large and/or diverse datasets over which complex aggregates cannot be successfully executed, and so forth. This again creates a practical limit with respect to how detailed a content description Sportal can generate for certain endpoints. For example, in later results we will show that while 94% of operational public endpoints respond successfully when asked for a list of all classes in their dataset, only 40% respond successfully when additionally asked how many instances those classes have. Thus, the Sportal catalogue would include metadata about the classes that appear in 93% of the catalogued endpoints, but only in 40% cases would the catalogue have information about how many instances appear in those classes.

Rather than limiting ourselves to building uniform descriptions of each endpoint based on information that can be computed from, say, >90% of endpoints, Sportal also considers more complex queries in its scope: while most endpoints cannot return responses to such queries, as we will show, a non-trivial percentage of endpoints do respond. In the interest of collecting as much data as possible from these latter endpoints, we include these more complex queries. Likewise we would hope that as SPARQL implementations mature, the percentage of endpoints responding to more complex queries may grow over time. As a result, the descriptive metadata available for an individual endpoint may differ from others depending on its ability to answer increasingly complex queries over its dataset. A core contribution of this paper is thus to evaluate the ability of public SPARQL endpoints to answer increasingly complex self-descriptive queries, which reflects the coverage of high-level metadata available to the Sportal catalogue (and similar agents) using only the SPARQL interface.

More specifically, our working hypothesis in this paper is that – despite problems with endpoint reliability and performance – by computing content descriptions using self-descriptive queries issued directly to endpoints, we can create a catalogue with (i) broader coverage and (ii) more up-to-date information than existing catalogues of SPARQL endpoints that rely on currently-available content descriptions provided by the publishers themselves. Towards investigating the validity of this hypothesis, this paper is structured as follows:

- We begin in Section 2 with some background on related areas: Linked Data access methods, proposals for describing RDF datasets, proposals for schemes to help find relevant SPARQL endpoints, as well as discussion on how the problem could be viewed from the perspective of Linked Data as a Distributed System.

- In Section 3, we look at how the content of endpoints can be described in a general-purpose, automated manner. In order to extract a *description* of the content of each endpoint, we propose to use a set of 29 self-descriptive SPARQL (1.1) queries that capture a large "computable" subset of a VoID description [4] as well as some additional features.[2]

- In Section 4, we first investigate, in a controlled environment, how well current SPARQL 1.1 engines are able to process these self-describing queries, some of which involve aggregation across an entire dataset and thus may require a prohibitive amount of processing, especially for large datasets. We run experiments over four datasets of increasing size and complexity using four SPARQL engines – 4store [18], Jena/Fuseki[3], Sesame [9] and Virtuoso [14] – to see how well our self-descriptive queries perform. These engines are mostly commonly used to power public SPARQL endpoints [10].

- With an idea of how the queries perform in a local environment for a variety of datasets and engines, in Section 5, we investigate how effectively public SPARQL endpoints process these queries. We take a list of 618 public endpoints and investigate the ratio that can answer each of the

[2]Certain aspects of VoID may not be computable directly from a dataset, such as the author(s) of a dataset, how it is licensed, OpenSearch descriptions, etc. Likewise we do not include subjective criteria in the computable fragment – such as the categories of the dataset – even if candidates could be computed automatically [15].

[3]http://jena.apache.org/documentation/fuseki2/; l.a. 2015/12/10

self-descriptive queries and characterise the typical performance we can expect in a realistic, uncontrolled environment. Our results show that, depending on the query, the ratio of operational endpoints[4] returning non-empty (but possibly partial) responses varies from 25–94%.

- In Section 6, we introduce the Sportal catalogue based on the results collected from the remote endpoints. We describe the manner in which it can help both human and software agents to find public endpoints on the Web that may be relevant for their needs. Based on the results of the previous questions, we discussed the (in)completeness of the catalogue and both the capabilities and limitations of the system. We also provide a high-level comparison of the Sportal catalogue with two catalogues based on publisher-provided content descriptions: VoID Store and DataHub.
- We conclude in Section 7 by recapitulating the main results of the paper and lessons learnt with respect to the goal of building a central catalogue of public SPARQL endpoints.

2 Background

Before we continue to the core of the paper, we provide some brief background on (1) methods for accessing Linked Data, (2) the problem of peer discovery in the area of Distributed Systems, (3) works on finding relevant SPARQL endpoints, and (4) techniques for describing/summarising RDF datasets.

Linked Data access methods

Traditionally there have been three methods provided for consumer agents to access content from knowledge-bases published as Linked Data: dereferencing, where IRIs of interest are looked up via HTTP; dumps, where the entire content of a dataset is made available for download; and SPARQL endpoints, where a query interface is provided over the local content. A more recent proposal – Linked Data Fragments [46] – has recently begun to gain attention.

Both dereferencing and dumps are lightweight methods in-tune with current practices on the Web; however, they can be inefficient for agents to use. Consider an agent wishing to retrieve the populations of Asian capitals from DBpedia. An agent has no direct way of finding the correct IRIs to dereference; even if they did, DBpedia specifies a `Crawl-delay` of 10 seconds: assuming that the DBpedia IRIs of 49 Asian capitals needed dereferencing, a polite agent would require 8 minutes to retrieve the respective documents and would ultimately use one triple out of potentially hundreds of thousands in each document. Using a dump would entail downloading an entire dataset to get at 49 triples; hosting a local dump mirror would require constant refreshing.

Hence publishers provide SPARQL endpoints as a convenient alternative to dereferencing or dumps. To get the populations of Asian capitals, an agent could run the following query against the DBpedia SPARQL endpoint[5]:

```
SELECT ?pop ?city WHERE { ?city dct:subject dbc:Capitals_in_Asia ; dbo:populationTotal ?pop . }
```

All going well, the query will return populations in less than a second. Likewise only the data that the client is interested in will be transferred. However, SPARQL endpoints push the burden from data consumers to producers: hosting such a public query service is expensive and as a result, endpoints may not be able to answer all queries for all consumer agents [10]. Despite problems with reliability, SPARQL endpoints still offer an appealing method for consumer agents to interact with remote Linked Data knowledge-bases where endpoints such as DBpedia serve millions of queries for clients [16].

As an alternative to SPARQL endpoints, Verborgh et al. [46] propose methods for providing and organising multiple access methods to a Linked Dataset, including a lightweight "triple pattern fragment", which allows clients to request all triples matching a single pattern, the goal of which is to allow publishers to host highly reliable but greatly simplified query services, thus trying to strike a better balance between the costs on the client and server side. Although their Linked Data Fragments (LDF) proposal offers a valuable compromise between client and server costs, being a recent proposal, SPARQL endpoints still greatly outnumber the number of LDF servers on the Web.

[4]We say that an endpoint is *operational* if it can be accessed over HTTP through the SPARQL protocol and will return a valid non-empty response to the following query: `SELECT * WHERE ?s ?p ?o  LIMIT 1`

[5]http://dbpedia.org/sparql; l.a. 2015/12/10 (42 populations are returned at the time of writing).

SPARQL Endpoints as a Distributed System

Viewed from the perspective of Distributed Computing, each SPARQL endpoint on the Web involves a client–server architecture, where numerous clients use the SPARQL protocol to interface with a single external server.[6] However, when hundreds of public SPARQL endpoints are viewed collectively, they can be seen as forming a decentralised peer-to-peer (P2P) system. In particular, with the advent of SPARQL 1.1 Federation [37], endpoints can query each other and thus may perform computation on behalf of other peers.

In this light, the goal of finding relevant SPARQL endpoints relates to the core problem of peer discovery in the P2P area, wherein a peer wishes to find another peer with a particular piece of data. To make this task more efficient, *structured P2P systems* impose an overall organisation on the network overlay to ensure rapid peer discovery; the most common structure is a Distributed Hash Table (DHT), which is effectively a distributed map where keys are hashed to determine on which peer(s) a given set of key–value pairs should be stored [42, 48, 40, 44]. However, all such structured schemes assume that peers in the network can be assigned data, which is not true of SPARQL endpoints where peers themselves decide which datasets they wish to index.

As such, public SPARQL endpoints collectively form an *unstructured P2P system*, where, since there is no correlation imposed between a peer and the data it indexes, peer discovery would necessarily involve one of two options: a separate search index that records the content at each peer (e.g., trackers in BitTorrent [38]), or blindly flooding the network with queries looking for the desired data from peers in a "brute force" manner (e.g., Gnutella [41]).

Rather than requiring a complete global structure or accepting zero structure, other proposals aim to strike a balance by imposing a limited form of structure over nodes. For example, routing indices [12] allow nodes to index whatever data they wish, but require that each peer must additionally store pointers to a neighbouring peer that is closer to the desired data; this avoids blind flooding of queries during peer discovery, instead allowing peers to be *routed* to relevant peer(s). Likewise, routing indexes avoid the need for a central index of peer content.

However, our goal in this paper is to enable peer discovery of SPARQL endpoints without changing the current infrastructure; we feel that it is important to explore options over the current infrastructure first before proposing that hundreds of stakeholders change how they host their data. For example, we do not presume that publishers will agree to add and maintain routing indexes towards the endpoints of external publishers. Hence we assume that no structure is imposed on the peers, but rather that each SPARQL endpoint indexes its own data. Thus the scenario is effectively unstructured: we have no guarantees about which data may appear at which endpoint/peer. Our hypothesis instead is that we can use the SPARQL query interface to learn about the content at each peer.

Describing/Summarising RDF datasets

With respect to building a central search service for endpoints based on their content, it would seem infeasible to index all of the data from the endpoint, hence some form of summary or schema overview must be indexed. A variety of works have proposed methods to describe and/or summarise RDF datasets.

In terms of describing metadata about RDF datasets, Cyganiak et al. [13] propose Semantic Sitemaps to mark the locations of different Linked Data access points; however information captured is limited to broad concepts such as change frequency. Alexander et al. [4] later proposed VoID for describing RDF datasets and the links between them. As we will see, the vocabulary provides terms for describing high-level statistics about a dataset, as well as about the instances of specific classes and the usage of specific properties. A number of works have proposed extensions to the VoID vocabulary. Mountantonakis et al. [34] propose to extend VoID with metrics about the connectivity of pairs of data sources to capture, for example, the number of common RDF terms used in both sources, the increase in average node degree with both sources are combined, etc. Omitola et al. [35] propose to extend VoID to allow publishers to describe in more depth the provenance of their dataset.

With respect to computing dataset descriptions, or profiling datasets, Bohm et al. [8] demonstrated that computing a VoID description for large datasets is feasible using MapReduce techniques. As part of the LOD Laundromat service – which aims to clean up and republish existing datasets in a more uniform

[6]The single server itself of course may be a distributed system, involving multiple replicated or clustered machines [18, 21]; however, this is all transparent from the perspective of the client, who sees one server.

manner – Beek et al. [7] compute a VoID description for each dataset indexed. More recently, Fetahu et al. [15] propose extracting topics from a dataset based on a combination of information retrieval techniques such as PageRank, HITS and Named Entity Recognition applied offline over the dataset. Abejan et al. [1] propose ProLOD++: a system to profile Linked Datasets that applies clustering techniques, statistical analysis, and association rules to find semantically related groups of entities, statistical distributions, properties that together uniquely identify resources, as well as suggested changes to the dataset/ontology. Mihindukulasooriya et al. [33] propose Loupe: a system that extracts a schema-level summary of a dataset similar to that captured by VoID (e.g., number of triples, number of instances per class, etc.), with additional information about namespaces, ontological definitions, etc.

Closer to our own contribution, various works have proposed using SPARQL to extract high-level information about an RDF dataset. Auer et al. [5] propose LODstats, which applies analytics over a stream of RDF data but which uses SPARQL filters to (reject)/select (ir)relevant triples; use of SPARQL is limited to filters. Langegger & Wöß propose RDFStats [28], which uses a pipeline of SPARQL (1.0) queries to generate a histogram on a per-class basis, representing the predicates and types of values associated with its instances. Holst & Höfig [23] propose the use of SPARQL 1.1 queries to discover specific aspects of an RDF dataset, but the authors do not consider VoID and only run local experiments over three datasets. Mountantonakis et al. [34] propose a set of SPARQL 1.1 queries that can compute the connectivity metrics with which they extend VoID. Mäkelä [31] propose Aether: a system for extracting extended VoID descriptions from a SPARQL 1.1 compliant endpoint; this work is perhaps most similar in spirit to ours, however, the focus is more on getting an overview of a known endpoint rather than building a catalogue that can be used by clients to find endpoints of interest.

The SPARQL 1.1 Service Description (SD) [47] vocabulary was recently recommended by the W3C; however, unlike previously discussed works, which focus on describing the content of datasets, SD describes technical aspects of an endpoint, such as features supported, dataset configurations, etc.

Other works have focused on summarising the content of RDF datasets (rather than describing them using a high-level RDF description). Umbrich et al. [45] propose to use an approximate, hash-based indexing structure, called a QTree, to aid in source selection; the QTree allows for determining which sources are likely to contain matches for a given RDF triple pattern but at a fraction of the size of the original dataset. Khatchadourian & Consens [26] propose creating bisimulation labels that capture connectivity in an RDF graph on the level of the namespaces of the instance URIs and the schema used. Campinas et al. [11] propose using existing graph summary algorithms to summarise RDF graphs, where nodes that are equivalent per some relation – e.g., having the same types, or the same attributes – are collapsed into a single node to create a smaller summary graph.

Discovering SPARQL endpoints

As previously discussed in the introduction, there are two high-level options for discovering SPARQL endpoints with relevant data: (1) flood the endpoints with queries, or (2) build a central search index. For example, federated SPARQL engines employ one or both of these strategies [39, 43, 3, 2, 6]. Our goal is to build a central catalogue based on data collected from endpoints through their SPARQL interfaces.

Paulheim & Hertling [36] looked at how to find a SPARQL endpoint containing content about a given Linked Data URI: using VoID descriptions and the DataHub catalogue, the authors could find suitable endpoints for about 15% of the sample of ten thousand URIs considered. Mehdi et al. [32] looked at the problem of discovering endpoints that may be relevant to a set of domain-specific keywords: their approach involved generating a list of RDF literals from the keywords and flooding queries against endpoints to see if they contained, e.g., case or language-tag variations of the literals.

Buil-Aranda et al. [10] propose SPARQLES as a catalogue of SPARQL endpoints, but focus on performance and stability metrics rather than cataloguing content; they do however remark that they could only find static descriptions for the content of about one third of the public endpoints surveyed, making endpoint discovery difficult. Likewise, the analysis by Lorey [30] of public endpoints focused on characterising the performance offered by these services rather than on the problem of discovery.

There are a variety of locations online where lists of public endpoints can be found and searched over. For example, DataHub[7] provides a list of hundreds of Linked Datasets, which can be filtered to find those that offer SPARQL endpoint locations. One can, for example, search for datasets relating to

[7]http://datahub.io/, l.a. 2015/12/10.

"`uk crime`" and filter to only show those with SPARQL endpoints. However, the search functionality provided is limited in most cases to keyword search over the dataset title, or to browsing datasets with a given tag, etc. Still, the service often provides links to VoID files that could be used to catalogue the content of endpoints. Unlike SPORTAL however, these VoID files are provided by publishers rather than being computed from the endpoints. Hence we will later compare our catalogue with that formed by collecting the VoID files that DATAHUB links to for each dataset.

As part of the RKBexplorer infrastructure [17], the VOID STORE allows for performing searches over VoID files submitted to the system.[8] A service is also provided to find endpoints that index content about a given resource (using the `REGEX` patterns sometimes provided in VoID). This catalogue could thus be used by clients to find relevant SPARQL endpoints. Currently the store contains information related to 118 endpoints. Like DATAHUB – but unlike SPORTAL – the VoID files indexed by VOID STORE are again computed and uploaded by publishers.

Novelty

We focus on the problem of helping clients find relevant SPARQL endpoints. To the best of our knowledge, there are two online services that clients could use to try to find SPARQL endpoints based on their content: DATAHUB and/or VOID STORE. However, both of these services rely on static content descriptions provided by publishers themselves. As noted by Buil-Aranda et al. [10], many of the endpoints listed in the DATAHUB have been offline for years; also, of the endpoints surveyed, VoID descriptions are only available in the DATAHUB for 33.3% and in the VOID STORE for 22.4%. We instead propose to compute extended VoID descriptions for public endpoints directly through their SPARQL interface. We provide a high-level comparison between SPORTAL, DATAHUB and VOID STORE in Section 6.

3 Self-Descriptive Queries

With respect to describing the content of an endpoint, in this section, we list the set of SPARQL 1.1 queries that we use to compute a VoID-like description from the content indexed by an endpoint.

Functionality

First we wish to filter unavailable endpoints and to determine those that (partially) support SPARQL 1.1. We consider an endpoint available if it is accessible through the HTTP SPARQL protocol, it responds to a SPARQL-compliant query, and it returns a response in an appropriate SPARQL format; for this, we use query Q_{A1} (see Table 1), which should be trivial for an endpoint to compute, returning a single binding for any triple. We consider an endpoint SPARQL 1.1 aware if it likewise responds to a query valid only in SPARQL 1.1; for this, we use query Q_{A2} (see Table 1), which tests two features unique to SPARQL 1.1: sub-queries and the count aggregate function.[9]

Table 1: Queries for basic functionality

№	**Query**
Q_{A1}	`SELECT * WHERE { ?s ?p ?o } LIMIT 1`
Q_{A2}	`SELECT (COUNT(*) as ?c) WHERE { SELECT * WHERE { ?s ?p ?o } LIMIT 1 }`

Dataset-level statistics

Second we list a set of queries to capture high-level "dataset-level" statistics that form a core part of VoID. We issue five queries, as listed in Table 2, to ascertain the number of triples (Q_{B1}), and the number of distinct classes (Q_{B2}), properties (Q_{B3}), subjects (Q_{B4}) and objects (Q_{B5}). These queries require support for SPARQL 1.1 `COUNT` and sub-query features (as tested in Q_{A2}). The `<D>` term refers to an IRI constructed from the SPARQL endpoint's URL to indicate the dataset it indexes.

[8] `http://void.rkbexplorer.com/`; l.a. 2015/12/10

[9] This does not imply that the endpoint is fully compliant with SPARQL 1.1; only that it supports a subset of features.

Table 2: Queries for dataset-level VoID statistics

№	Query
Q_{B1}	`CONSTRUCT {  <D> v:triples ?x } WHERE { SELECT (COUNT(*) AS ?x) WHERE { ?s ?p ?o } }`
Q_{B2}	`CONSTRUCT { <D> v:classes ?x } WHERE { SELECT (COUNT(DISTINCT ?o) AS ?x) WHERE { ?s a ?o } }`
Q_{B3}	`CONSTRUCT { <D> v:properties ?x } WHERE { SELECT (COUNT(DISTINCT ?p) AS ?x) WHERE { ?s ?p ?o } }`
Q_{B4}	`CONSTRUCT { <D> v:distinctSubjects ?x } WHERE { SELECT (COUNT(DISTINCT ?s) AS ?x) WHERE { ?s ?p ?o } }`
Q_{B5}	`CONSTRUCT { <D> v:distinctObjects ?x } WHERE { SELECT (COUNT(DISTINCT ?o) AS ?x) WHERE { ?s ?p ?o } }`

Once these statistics are catalogued for public SPARQL endpoints, agents can use them to find endpoints indexing datasets that fall within a given range of triples in terms of overall size, or, for example, to find the endpoints with the largest datasets. Counts may be particularly useful – in combination with later categories – to order the endpoints; for example, to find the endpoints with a given class (using data from the next category) and order them by the total number of triples they index.

Class-based statistics

Third we ascertain similar statistics about the instances of each class following the notion of *class partitions* in VoID: a subset of the data considering only triples where instances of that class are in the subject position. Table 3 lists the six queries we use. The first query (Q_{C1}) merely lists all class partitions. The other five queries ($Q_{C2–6}$) count the triples and distinct classes, predicates, subjects and objects for each class partition; e.g., Q_{C2} retrieves the number of triples where instances of that class are in the subject position. Queries $Q_{C2–6}$ introduce `COUNT`, sub-queries and also `GROUP BY` features from SPARQL 1.1.

Table 3: Queries for statistics about classes

№	Query
Q_{C1}	`CONSTRUCT { <D> v:classPartition [ v:class ?c ] } WHERE { ?s a ?c }`
Q_{C2}	`CONSTRUCT { <D> v:classPartition [ v:class ?c ; v:triples ?x ] } WHERE { SELECT (COUNT(?p) AS ?x) ?c WHERE { ?s a ?c ; ?p ?o } GROUP BY ?c }`
Q_{C3}	`CONSTRUCT { <D> v:classPartition [ v:class ?c ; v:classes ?x ] } WHERE { SELECT (COUNT(DISTINCT ?d) AS ?x) ?c WHERE { ?s a ?c , ?d } GROUP BY ?c  }`
Q_{C4}	`CONSTRUCT { <D> v:classPartition [ v:class ?c ; v:properties ?x ] } WHERE { SELECT (COUNT(DISTINCT ?p) AS ?x) ?c WHERE { ?s a ?c ; ?p ?o } GROUP BY ?c }`
Q_{C5}	`CONSTRUCT { <D> v:classPartition [ v:class ?c ; v:distinctSubjects ?x ] } WHERE { SELECT (COUNT(DISTINCT ?s) AS ?x) ?c WHERE { ?s a ?c } GROUP BY ?c }`
Q_{C6}	`CONSTRUCT { <D> v:classPartition [ v:class ?c ; v:distinctObjects ?x ] } WHERE { SELECT (COUNT(DISTINCT ?o) AS ?x) ?c WHERE { ?s a ?c ; ?p ?o } GROUP BY ?c  }`

Once catalogued, agents can use statistics describing class partitions of the datasets to find endpoints mentioning a given class, where they can additionally (for example) sort results in descending order according to the number of unique instances of that class, or triples used to define such instances, and so forth. Hence the counts computed by $Q_{C2–6}$ help agents to distinguish endpoints that may only have one or two instances of a class to those with thousands or millions. Likewise criteria can be combined arbitrarily for multiple classes, or with the overall statistics computed previously.

Property-based statistics

Fourth we look at property partitions in the dataset, where a property partition refers to the set of triples with that property term in the predicate position. Queries are listed in Table 4. As before, Q_{D1} lists the property partitions. $Q_{\text{D2–4}}$ count the number of triples, distinct subjects and distinct objects. We do not count classes (which would be 0 for all properties except `rdf:type`) or properties (which would always be 1).

Table 4: Queries for statistics about properties

№	Query
Q_{D1}	`CONSTRUCT { <D> v:propertyPartition [ v:property ?p ] } WHERE { ?s ?p ?o }`

Q_{D2}	`CONSTRUCT { <D> v:propertyPartition [ v:property ?p ; v:triples ?x ] }`
	`WHERE { SELECT (COUNT(?o) AS ?x) ?p  WHERE { ?s ?p ?o } GROUP BY ?p }`

Q_{D3}	`CONSTRUCT { <D> v:propertyPartition [ v:property ?p ; v:distinctSubjects ?x ] }`
	`WHERE { SELECT (COUNT(DISTINCT ?s) AS ?x) ?p WHERE { ?s ?p ?o } GROUP BY ?p }`

Q_{D4}	`CONSTRUCT { <D> v:propertyPartition [ v:property ?p ; v:distinctObjects ?x ] }`
	`WHERE { SELECT (COUNT(DISTINCT ?o) AS ?x) ?p WHERE { ?s ?p ?o } GROUP BY ?p }`

Using these statistics about property partitions in the catalogue, agents can, for example, retrieve a list of public endpoints using a given property ordered by the number of triples using that specific property. Likewise criteria can be combined arbitrarily for multiple properties, or with the dataset- or class-level metadata previously collected; for example, an agent may wish to order endpoints by the *ratio* of triples using a given property (where the count from Q_{D2} for the property in question can be divided by the total triple count from Q_{B1}), or to find endpoints where all subjects have an `rdfs:label` value (where the count computed from Q_{D3} for that property should match the count for Q_{B4}).

Nested class–property statistics

Fifth we look at how classes and properties are used together in a dataset, gathering statistics on property partitions nested within class partitions: these statistics detail how properties are used for instances of specific classes. Table 5 lists the four queries used. Q_{E1} lists the property partitions nested inside the class partitions, and $Q_{\text{E2–4}}$ count the number of triples using a given predicate for instances of that class, as well as the number of distinct subjects and objects those triples have. In terms of technical features, these queries involve `GROUP BY` over multiple terms. In general, the queries listed in this section are quite complex where we would expect that many endpoints would struggle to return metadata about their content at this detailed level of granularity.

Table 5: Queries for nested property/class statistics

№	Query

Q_{E1}	`CONSTRUCT { <D> v:classPartition [ v:class ?c ; v:propertyPartition [ v:property ?p ] ] }`
	`WHERE { ?s a ?c ; ?p ?o }`

Q_{E2}	`CONSTRUCT { <D> v:classPartition [ v:class ?c`
	`v:propertyPartition [ v:property ?p ; v:triples ?x ] ] }`
	`WHERE { SELECT (COUNT(?o) AS ?x) ?p  WHERE { ?s a ?c ; ?p ?o }  GROUP BY ?c ?p  }`

Q_{E3}	`CONSTRUCT { <D> v:classPartition [ v:class ?c ;`
	`v:propertyPartition [ v:distinctSubjects ?x ] ] }`
	`WHERE { SELECT (COUNT(DISTINCT ?s) AS ?x) ?c ?p WHERE { ?s a ?c ; ?p ?o } GROUP BY ?c ?p }`

Q_{E4}	`CONSTRUCT {  <D> v:classPartition [ v:class ?c ;`
	`v:propertyPartition [ v:distinctObjects ?x ; v:property ?p ] ] }`
	`WHERE { SELECT (COUNT(DISTINCT ?o) AS ?x) ?c ?p WHERE { ?s a ?c ; ?p ?o } GROUP BY ?c ?p }`

An agent could use the resulting metadata to find endpoints describing instances of specific classes with specific properties, with filtering or sorting criteria based on, e.g., the number of triples. For example, an agent might be specifically interested in images of people, where they would be looking for the class-partition `foaf:Person` with the nested property-partition `foaf:depicts`. Using the previous statistics, it would have been been possible to find endpoints that have data for the class `foaf:Person` and triples with the property `foaf:depicts`, but not that the images were defined for people. The counts from $Q_{E2–E4}$ again allow an agent to filter or order endpoints by the amount of relevant data.

Miscellaneous statistics

In our final set of experiments, we look at queries that yield statistics not supported by VoID as listed in Table 6. In particular, we experiment to see if endpoints can return a subset of statistics from the VoID Extension Vocabulary[10], which include counts of different types of unique RDF terms in different positions: subjects IRIs (Q_{F1}), subject blank nodes (Q_{F2}), objects IRIs (Q_{F3}), literals (Q_{F4}), object blank nodes (Q_{F5}), all blank nodes (Q_{F6}), all IRIs (Q_{F7}), and all terms (Q_{F8}). Inspired by the notion of "schema maps" as proposed by Kinsella et al. [27], we also count the classes that the subjects and objects of specific properties are instances of ($Q_{F9–10}$); these are "inverses" of queries ($Q_{E3–4}$).[11]

Table 6: Queries for miscellaneous statistics

№	Query
Q_{F1}	`CONSTRUCT { <D> e:distinctIRIReferenceSubjects ?x } WHERE { SELECT (COUNT(DISTINCT ?s ) AS ?x) WHERE { ?s ?p ?o  FILTER(isIri(?s))} }`
Q_{F2}	`CONSTRUCT { <D> e:distinctBlankNodeSubjects ?x } WHERE { SELECT (COUNT(DISTINCT ?s) AS ?x) WHERE { ?s ?p ?o  FILTER(isBlank(?s))} }`
Q_{F3}	`CONSTRUCT { <D> e:distinctIRIReferenceObjects ?x } WHERE { SELECT (COUNT(DISTINCT ?o ) AS ?x) WHERE {  ?s ?p ?o  FILTER(isIri(?o))} }`
Q_{F4}	`CONSTRUCT { <D> e:distinctLiterals ?x } WHERE { SELECT (COUNT(DISTINCT ?o ) AS ?x) WHERE { ?s ?p ?o  FILTER(isLiteral(?o))} }`
Q_{F5}	`CONSTRUCT { <D> e:distinctBlankNodeObjects ?x } WHERE { SELECT (COUNT(DISTINCT ?o ) AS ?x) WHERE { ?s ?p ?o  FILTER(isBlank(?o))} }`
Q_{F6}	`CONSTRUCT { <D> e:distinctBlankNodes ?x } WHERE { SELECT (COUNT(DISTINCT ?b ) AS ?x) WHERE { { ?s ?p ?b } UNION { ?b ?p ?o } FILTER(isBlank(?b)) } }`
Q_{F7}	`CONSTRUCT { <D> e:distinctIRIReferences ?x } WHERE { SELECT (COUNT(DISTINCT ?u ) AS ?x) WHERE { { ?u ?p ?o } UNION { ?s ?u ?o } UNION { ?s ?p ?u } FILTER(isIri(?u) } }`
Q_{F8}	`CONSTRUCT { <D> e:distinctRDFNodes ?x } WHERE { SELECT (COUNT(DISTINCT ?n ) AS ?x) WHERE { { ?n ?p ?o } UNION { ?s ?n ?o } UNION { ?s ?p ?n } } }`
Q_{F9}	`CONSTRUCT { <D> v:propertyPartition [ v:property ?p ; s:subjectTypes [ s:subjectClass ?sType ; s:distinctMembers ?x ] ] } WHERE { SELECT (COUNT(?s) AS ?x) ?p ?sType WHERE { ?s ?p ?o ; a ?sType . } GROUP BY ?p ?sType }`
Q_{F10}	`CONSTRUCT { <D> v:propertyPartition [ v:property ?p ; s:objectTypes [ s:objectClass ?oType ; s:distinctMembers ?x ] ] } WHERE { SELECT (COUNT(?o) AS ?x) ?p ?oType WHERE { ?s ?p ?o . ?o a ?oType . } GROUP BY ?p ?oType }`

The resulting data could serve a number of purposes for agents looking for public endpoints. For example, the agent in question could look for datasets without any blank nodes, or for datasets where a given number of the objects of a given property are of a certain type. Likewise, the user can combine these criteria with earlier criteria; for example, to find endpoints with more than ten million triples where at least 30% of the unique object terms are literals.

[10] `http://ldf.fi/void-ext#`; denoted herein as `e:`.
[11] We created a novel namespace (`s:`) available from `http://vocab.deri.ie/sad#`.

4 Local Experiments

In our first set of experiments, we test whether or not SPARQL implementations can locally answer the queries we specified in the previous section. These implementations are used to power individual endpoints and hence we would like to see if running these queries is feasible in a locally controlled environment before running remote experiments.

Along these lines, given the 29 self-descriptive queries mentioned previously, we test four popular SPARQL query engines: Virtuoso (07.10.3207), Fuseki (1.0.2), 4store (4s-httpd/v1.1.4) and Sesame (2.7.12). Given that the cost of the self-descriptive queries listed previously depends directly on the size and nature of the dataset indexed, for each engine, we perform experiments with respect to the four real-world datasets listed in Table 7, representing a mix of datasets at a variety of scales and with a variety of diversity in predicates and classes used. The experiments are run on a server with Ubuntu 14, a 4x Intel Core i5 CPU (M540@2.53 GHz) processor and 8 GB of RAM. A timeout of 10 minutes was set for the first result to return. Result-size thresholds were switched off where applicable.

Table 7: High-level statistics for datasets used in local experiments

	Triples	**Subjects**	**Predicates**	**Objects**	**Classes**
DrugBank	517,023	19,693	119	276,142	8
Jamendo	1,049,647	335,925	26	440,686	11
Kegg	1,090,830	34,260	21	939,258	4
DBPedia	114,456,676	11,194,893	53,200	27,518,753	447

All of the engines passed the functionality tests. In Table 8, we list the runtimes for all other queries (spanning Tables 2–6), for the four datasets and the four engines. We manually inspected the results so as to only include runtimes where the correct response was returned. With respect to the (partially) failed queries, in the table, we differentiate between:

empty results (—) where zero results are returned, most commonly caused by a 10 minute timeout;

partial results (∼) where the stream of results returned is correct but ends prematurely;

incorrect results (×) where the results returned are false, most commonly caused by counts not considering all results or by query processor bugs.

In the following, we draw high-level conclusions from these results.

Datasets

We see in Table 8 that while Fuseki, Sesame and Virtuoso successfully run almost all queries for DrugBank, Jamendo and KEGG – datasets around or below a million triples – all engines struggle for the DBpedia dataset, which is two orders of magnitude larger and contains two orders of magnitude more classes, predicates, objects and subjects. This is better illustrated by Figure 1, where the difference between DBpedia and the other datasets is evident in terms of success rate. Only Virtuoso managed to return correct results for some queries over DBpedia, including counts for triples, classes, properties, triples per property partition, blank node subjects, blank node objects, blank nodes in any position[12] and object IRIs. We posit that with the available memory, queries over the smaller datasets could be processed largely in-memory whereas queries over DBpedia may have led to a lot of on-disk processing.

Engines

With respect to the success rate of the four implementations, from Table 8 and Figure 1, we can see that 4store struggled the most with the self-descriptive queries specified, returning correct results only for counts of classes, properties and blank nodes. Fuseki was the most reliable engine for the smaller datasets, successfully answering all queries over DrugBank, Jamendo and KEGG, whereas Sesame and Virtuoso struggled on queries Q_{D1} and Q_{F10}. From further investigation, we discovered that for Q_{D1}

[12]In fact, DBpedia contained no blank nodes, nor did any of the other datasets.

Table 8: Local query runtimes for self-descriptive queries (times in millisecond; engines are keyed as **4**store, **F**useki, **S**esame, **V**irtuoso; '—' indicates empty results, '∼' partial results, '×' incorrect results)

№	DrugBank				Jamendo				KEGG				DBpedia			
	4	F	S	V	4	F	S	V	4	F	S	V	4	F	S	V
Q_{B1}	×	5,128	6,962	95	×	4,637	6,001	127	×	2,018	3,059	130	—	—	—	6,175
Q_{B2}	27	3,140	658	43	×	8,278	2,258	158	42	2,773	325	55	×	—	—	6,260
Q_{B3}	29	1,664	7,029	155	34	1,840	6,478	278	35	1,016	3,140	284	—	—	—	25,130
Q_{B4}	×	6,029	7,324	252	×	7,987	6,503	1,493	×	2,455	3,263	498	—	—	—	—
Q_{B5}	×	17,141	8,764	1,269	×	4,714	6,824	1,736	×	7,174	3,715	2,523	—	—	—	—
Q_{C1}	×	4,201	21,845	3,560	×	16,868	166,727	33,559	×	3,125	19,282	4,453	∼	—	—	—
Q_{C2}	×	5,342	9,305	374	×	5,336	10,086	291	×	3,097	5,609	360	—	—	—	—
Q_{C3}	×	949	758	99	×	2,946	5,153	390	×	801	526	119	—	—	—	—
Q_{C4}	×	2,844	9,423	949	×	3,940	10,028	1,135	×	1,990	5,933	1,110	—	—	—	—
Q_{C5}	×	227	479	178	×	991	2,952	2,255	×	249	258	226	—	—	—	—
Q_{C6}	×	17,612	10,393	2,979	×	5,530	10,339	3,472	×	3,483	6,802	4,301	—	—	—	—
Q_{D1}	×	80,119	754,030	—	×	106,486	—	—	×	35,902	323,709	—	—	—	—	—
Q_{D2}	×	13,468	8,787	109	×	7,675	740	90	×	5,022	3,917	111	—	—	—	41,188
Q_{D3}	×	2,718	90,936	1,425	×	3,870	8,322	3,105	×	1,580	4,120	1,764	—	—	—	—
Q_{D4}	×	15,504	10,252	1,710	×	4,983	8,355	2,298	×	2,258	4,356	3,298	—	—	—	—
Q_{E1}	×	60,310	1,296,659	17,899	×	39,836	2,663,655	34,926	×	17,825	683,787	37,204	—	—	—	—
Q_{E2}	×	13,644	9,410	445	×	5,086	12,886	279	×	2,947	6,469	240	—	—	—	—
Q_{E3}	×	4,113	10,029	2,815	×	6,246	10,522	3,464	×	3,505	7,021	2,763	—	—	—	—
Q_{E4}	×	20,489	10,603	3,415	×	6,449	10,649	3,448	×	3,359	7,756	4,596	—	—	—	—
Q_{F1}	×	2,587	9,990	428	×	4,181	9,131	1,943	×	1,375	4,622	822	—	—	—	—
Q_{F2}	30	3,066	8,992	52	36	2,407	8,178	42	45	1,410	4,151	54	—	—	—	378
Q_{F3}	×	17,559	11,069	486	×	3,663	9,416	1,467	×	1,711	4,470	1,815	—	—	—	65,398
Q_{F4}	×	15,862	9,580	1,100	×	3,061	9,775	644	×	1,761	4,592	1,160	—	—	—	—
Q_{F5}	29	14,686	8,877	122	40	3,678	7,755	162	38	1,652	3,880	168	—	—	—	16,192
Q_{F6}	52	17,208	17,820	130	58	4,914	15,968	173	60	2,049	7,691	204	—	—	—	16,323
Q_{F7}	×	23,360	31,675	1,333	×	11,497	27,428	4,183	×	4,456	13,401	3,131	—	—	—	—
Q_{F8}	×	26,517	27,651	1,788	×	10,565	23,549	6,728	×	5,122	11,642	3,615	—	—	—	—
Q_{F9}	×	4,725	9,546	414	×	6,746	9,875	315	×	3,210	6,822	271	—	—	—	—
Q_{F10}	×	1,272	—	127	×	5,290	—	619	×	1,228	—	149	—	—	—	—

– list all property partitions – the engines were returning two output triples for every triple indexed, producing a large volume of non-lean data, as opposed to returning two output triples for every unique property. To get around this, a sub-query specifying `DISTINCT` on `?p` could be used at the cost of requiring SPARQL 1.1 support. On the other hand, Q_{F10} would seem on face value to be the most expensive query in our collection, requiring an open join and aggregation step that may naturally fail even for small-to-medium-sized datasets.

Runtimes

We see quite a large variance in runtimes between the different engines, varying in orders of magnitude. In order to get a better insight into the differences in performance, in Figure 2, we plot the ratio of all 116 queries (29 queries $\times$ 4 datasets) that ran below a certain runtime, where, for example, we can see that Virtuoso successfully ran 40% of the queries in less than one second and 72% of the queries in less than ten seconds. The plots end where queries began to fail. Interestingly, although 4store was the most unreliable engine, it offered the fastest runtimes for the simpler queries it did answer, suggesting some index may have been used for optimisation purposes. Although Sesame and Fuseki were faster for certain queries, the trend in Figure 2 suggests that overall, Virtuoso was fastest for most queries. We also see that many queries continued to stream results well in excess of the one minute connection timeout, with Sesame having some of the slowest successful query executions (the slowest being 44 minutes).

Errors

Although all engines returned empty results, 4store was the only engine that was found to return partial or incorrect answers where, for count queries, the engine seemed to return a partial count of what it had

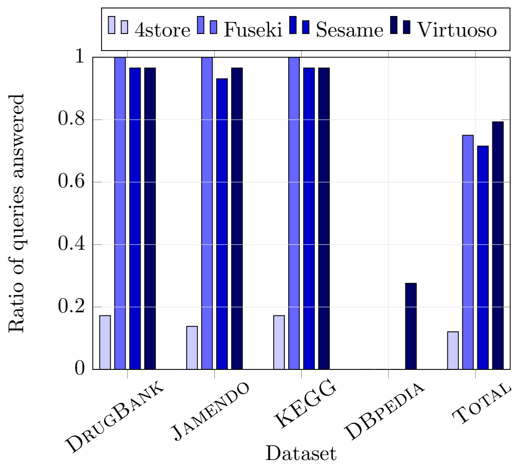

Figure 1: Ratio of successful queries per dataset/engine

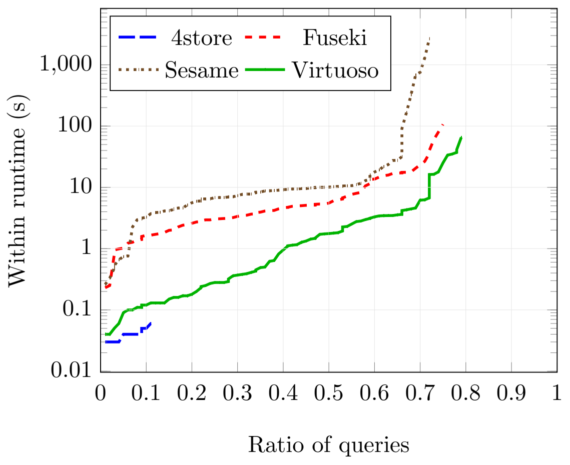

Figure 2: Ratio of queries executed within given runtime

found up to a certain point.[13] Otherwise, the other engines tended to have fail-stop errors, meaning that they either returned full correct results or no results at all. With respect to public SPARQL endpoints, although we would expect partial results due to result-size limits [10], the pattern of errors in Table 8 suggests that *if* one of the latter three engines successfully returns a count result, then that value is likely to be correct.

Summary

In general, we see that Fuseki, Sesame and Virtuoso are capable of describing – in considerable detail – the content of small-to-medium-sized datasets under controlled conditions, returning correct results for almost all queries over DrugBank, Jamendo and KEGG. These results suggest that when deployed as public SPARQL endpoints, these implementations could provide a rich catalogue of the content of such datasets. However, we would not expect to derive as rich or as trustworthy a description from 4store-powered endpoints, nor from larger datasets or datasets with more diverse schema terms.

5 Remote Experiments

We now look at how public SPARQL endpoints themselves perform for the list of self-descriptive queries we have previously enumerated. Along these lines, we collected a list of 540 SPARQL endpoints registered in the DataHub in April 2015.[14] We likewise collected a list of 137 endpoints from Bio2RDF releases 1–3. In total, we considered 618 unique endpoints (59 endpoints were present in both lists). The results are based on experiments we performed in April 2015.

Implementations used

With respect to the previous local experiments, we are first interested to see if we can determine which implementations are used by the in-scope endpoints. As per the observations of Buil-Aranda et al. [10], although there is no generic exact method of determining the engine powering a SPARQL endpoint, the HTTP header may contain some clues in the `Server` field. Hence our first step was to perform a lookup on the endpoint URLs. In Table 9, we present the response codes of this step, where we see that quite a large number of endpoints return error codes `4xx`, `5xx`, or some other exception. This indicates that a non-trivial fraction of the endpoints from our list are offline; we will return to this issue later.

[13]Many counts had the value of 1,996 or some other value close to a multiple of a thousand; these results were incorrect where some of the expected values were in the hundreds of thousands.

[14]`http://datahub.io`

Table 9: HTTP response

Response	**№**
200 (*successful*)	307
200 (*unsuccessful*)	43
400	56
404	66
500	4
502	0
503	32
unknown host	51
time out	23
connection refused	18
not responding	7

Table 10: Server Names

Server-field	**№**
`Apache`	203
`Virtuoso`	174
`nginx`	38
`Jetty`	25
`Fuseki`	15
`GlassFish`	3
`4s-httpd`	2
`lighttpd`	1
empty	130

With respect to the server names returned by those URLs that returned a HTTP response, Table 10 enumerates the main prefixes that we discovered. Although some of the server names denote generic HTTP servers – more specifically `Apache`, `nginx`, `Jetty`, `GlassFish`, `Restlet` and `lighttpd` – we also see some names that indicate SPARQL implementations – namely `Virtuoso`, `Fuseki` and `4s-httpd` (4store). Interestingly, we see that two of the engines that performed quite well in our local experiments – `Virtuoso` and `Fuseki` – are quite prevalent amongst SPARQL endpoints.[15]

Availability and version

Based on the previous experiment, we suspect some of the endpoints in our list may be offline. Hence we next look at how many endpoints respond to the basic availability query Q_{A1}.

Given that we run queries in an uncontrolled environment, we perform multiple runs to help mitigate temporary errors and remote server loads: the core idea is that if an endpoint fails at a given moment of time, a catalogue could simply reuse the most recent successful result. Along these lines, we ran three weekly experiments in the month of April 2015. In total, 307 endpoints (49.7%) responded to Q_{A1} at least once in the three weeks; we deem these endpoints to be *operational* and others to be *offline*. Of the operational endpoints, 7 (1.1%) responded successfully exactly once to Q_{A1}, 28 (4.5%) responded successfully exactly twice, and 272 (44.0%) responded successfully thrice. In the most recent run, 298 endpoints responded to Q_{A1}. Of these, 168 (56.4%) also responded with a single result for Q_{A2}, indicating some support for SPARQL 1.1 in about half of the operational endpoints.

Moving forward, to mitigate the issue of temporary errors, for each endpoint, we consider the most recent non-empty results returned for each endpoint and each query over the three runs.

Success rates

We first focus on the overall success rates for each query, looking at the ratio of the 307 endpoints that return non-empty results. The results are illustrated in Figure 3, where we see success rates varying from 25% for Q_{E3} on the lower end, to 94% for Q_{C1} on the higher end. The three queries with the highest success rates require only SPARQL 1.0 features to run: list all class partitions (Q_{C1}), all property partitions (Q_{D1}), and all nested partitions (Q_{E1}). Hence we see that – as expected given that only 49% could respond to the SPARQL 1.1 test query Q_{A2} – more endpoints can answer queries not requiring novel SPARQL 1.1 features such as counts or sub-queries. The query with the highest success rate that involved SPARQL 1.1 features was Q_{B1}, where 51% of endpoints responded with a count of triples. In general, queries deriving counts within partitions had the lowest success rates.

Result sizes

Next we focus on the size of results returned for each query. To illustrate this, in Figure 4 we show result sizes in log scale for individual queries at various percentiles considering all endpoints that returned a non-empty result. As expected, queries that return a single count triple return one result across all

[15]These results correspond quite closely with those of Buil-Aranda et al. [10]. We believe that some Sesame endpoints may be within the `Apache` category since the default Sesame header is `Apache-Coyote/1.1`.

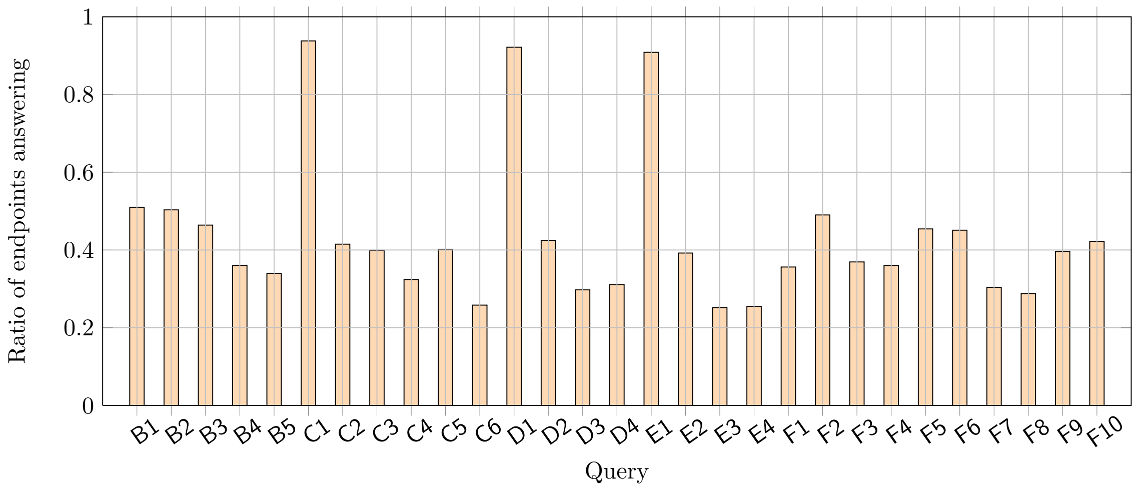

Figure 3: Ratio of endpoints returning non-empty results per query

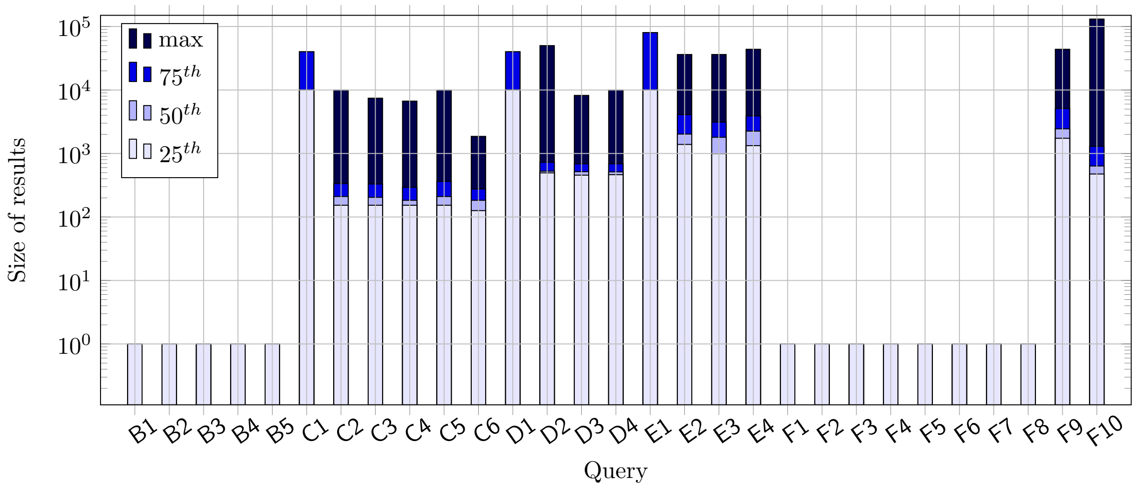

Figure 4: Sizes of results for different queries taking 25th, 50th (median), 75th and 100th (max) percentiles, inclusive, across all endpoints returning non-empty results

percentiles. For other queries, the result sizes extended into the tens of thousands. One may note that the higher percentiles are quite compressed for certain queries, indicating the presence of result thresholds. For example, for Q_{C1}, a common result-size was precisely 40,000, which would appear to be the effect of a result-size threshold. Hence, unlike the local experiments where result thresholds could be switched off, we see that for public endpoints, partial results are sometimes returned.

Runtimes

Finally we focus on runtimes for successfully executed queries, incorporating the total response time for issuing the query and streaming all results. In Figure 5, we again present the runtimes for each query considering different percentiles across all endpoints returning non-empty results in log scale. We see quite a large variance in runtimes, which is to be expected given that different endpoints host datasets of a variety of sizes and schemata on servers with a variety of computational capacity. In general, we see that the 25th percentile roughly corresponds with the one second line, but that slower endpoints may take tens or hundreds of seconds. The flat max trend seems to be the effect of remote timeout policies, where query runtimes often maxed out at between 100–120 seconds, likely returning partial results.

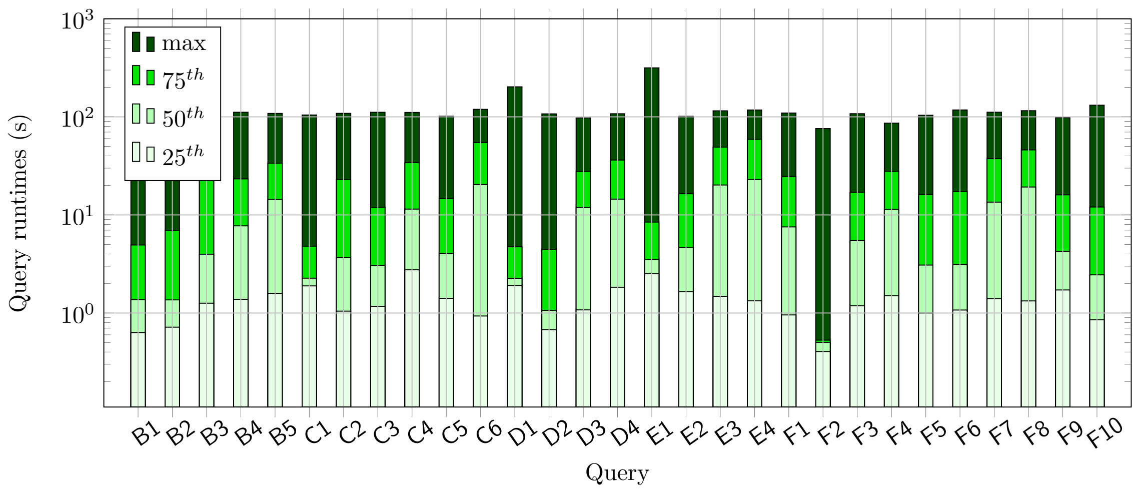

Figure 5: Runtimes for different queries taking 25th, 50th (median), 75th and 100th (max) percentiles, inclusive, across all endpoints returning non-empty results

Summary

Although we see a high success rate in asking for class and property partitions where we would expect to have such data for over 90% of the endpoints, the success rate for queries using novel SPARQL 1.1 features drops to 25–50%. We also noted that for queries generating larger result sizes, thresholds and timeouts would likely lead to only partial results being returned. But based on local experiments and the implementations most prominently used by endpoints, we posit that the partial data returned by these endpoints is likely to be accurate even if incomplete.

6 SPARQL Portal

Our primary motivation in this paper is to investigate a method for cataloguing the content of public SPARQL endpoints without requiring them to publish separate, static descriptions of their content—or indeed, for publishers to offer any additional infrastructure other than the query interface itself. In the previous sections, we performed a variety of experiments that characterised the feasibilities and limitations of collecting metadata about the content of endpoints by directly querying them. In this section, we describe the SPORTAL catalogue itself, including its interfaces, capabilities and limitations. A prototype of SPORTAL is available online at `http://www.sportalproject.org`.

Building the catalogue

The results of the self-descriptive queries are used to form a content description for each endpoint, which collectively form the SPORTAL catalogue. This catalogue is indexed in a local SPARQL endpoint that agents can access. The result for each self-descriptive query over each endpoint is loaded into a dedicated Named Graph and annotated with provenance information using the model illustrated in Figure 6, which follows the recommendations of the W3C PROV-O ontology [29] (based on the notion of *activities* and *entities*). Each query run is (implicitly)[16] considered to be an *activity*, with an associated *start time* and *end time*. This activity *uses* a query *entity* and an endpoint *entity* to *generate* a query-result *entity* (a Named Graph with the results). Each VoID dataset is *derived from* potentially multiple query-results. We also keep track of other information, such as HTTP response codes, the number of triples generated by the query, whether the query is SPARQL 1.0 or SPARQL 1.1, the text of the query, etc.

Example 1 provides a real-world example output from executing Q_{B1} over an endpoint, with provenance information following the model previously described.

[16] We do not explicitly type our entities with PROV-O classes simply to keep the data concise: memberships of the respective classes could be inferred from the domain/range of the PROV-O properties we use.

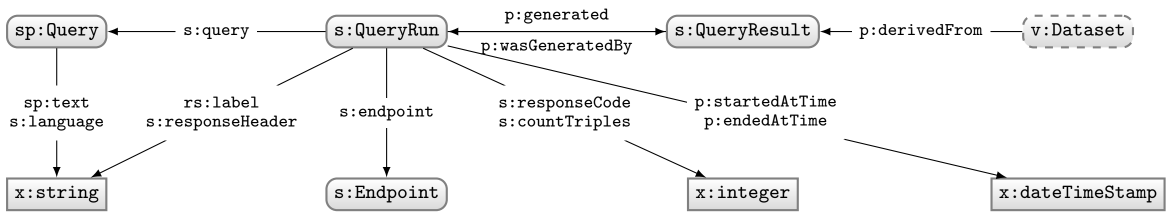

Figure 6: SPORTAL provenance data model

```
@prefix ep: <http://www.linklion.org:8890/sparql#> .
@prefix p: <http://www.w3.org/ns/prov#> .
@prefix s: <http://vocab.deri.ie/sad#> .
@prefix sp: <http://spinrdf.org/spin#> .
@prefix rs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix v: <http://rdfs.org/ns/void#> .
@prefix x: <http://www.w3.org/2001/XMLSchema#> .

## PROVENANCE metadata
# each query run is implicitly a p:Activity
 ep:totalNumberOfTriplesQueryRun a s:QueryRun ;
   p:generated ep:totalNumberOfTriplesResult ;
   s:responseCode 200 ; s:countTriples 1 ;
   p:startedAtTime "2015-05-11T21:20:54.065Z"^^x:dateTimeStamp ;
   p:endedAtTime "2015-05-11T21:20:55.511Z"^^x:dateTimeStamp ;
   rs:label "Extracting triple count from 'http://www.linklion.org:8890/sparql' on 2015-05-11T21:20:54.065Z"@en ;
   s:resultDataset ep:dataset ;
   s:responseHeader "StatusCode=[HTTP/1.1 200 OK] & Server=[Virtuoso/07.00.3203 (Linux) x86_64-suse-linux-gnu]" ;
   s:endpoint <http://www.linklion.org:8890/sparql> ; # s:endpoint sub-property of p:used
   s:query ep:totalNumberOfTriplesQuery . # s:query sub-property of p:used

# each query is implicitly a p:Entity
 ep:totalNumberOfTriplesQuery a sp:Query ;
   sp:text """PREFIX void: <http://rdfs.org/ns/void#>
              CONSTRUCT { <http://www.linklion.org:8890/sparql#dataset> void:triples ?count }
                WHERE { SELECT (COUNT(*) AS ?count) WHERE { ?s ?p ?o } }""" ;
   s:language "SPARQL1.1" .

# represents the RDF graph returned by the query, implicitly a p:Entity
 ep:totalNumberOfTriplesResult a s:QueryResult ;
   p:wasGeneratedBy ep:totalNumberOfTriplesQueryRun .

# connects the dataset mentioned in the results and the graph storing the results
 ep:dataset p:wasDerivedFrom s:totalNumberOfTriplesResult .

## RDF GENERATED BY THE QUERY
# loaded into the Named Graph ep:totalNumberOfTriplesResult
 ep:dataset v:triples 77455301 .
```

Example 1: An example RDF output for Q_{B1} with provenance metadata

SPARQL interface

Sportal itself provides a public SPARQL endpoint, where the RDF triples produced by the `CONSTRUCT` clauses of the self-descriptive queries issued against public endpoints can themselves be queried. This allows users with specific requirements in mind to interrogate our catalogue in a flexible manner.

To take a first example, a client could pose the following query asking for the SPARQL endpoints for which the catalogue has the top 5 largest triple counts:

```
SELECT DISTINCT ?endpoint ?triples
WHERE { ?dataset v:triples ?triples ; v:sparqlEndpoint ?endpoint . }
ORDER BY DESC(?triples) LIMIT 5
```

This will return the following answer:[17]

?endpoint	?triples
http://commons.dbpedia.org/sparql	1,229,690,546
http://lod.b3kat.de/sparql	981,672,146
http://www.linklion.org:8890/sparql	727,421,750
http://live.dbpedia.org/sparql	560,701,025
http://linked.opendata.cz/sparql	555,666,667

As another example, referring back to the second client scenario mentioned in the introduction, take a user who is interested in data about proteins and asks for endpoints with at least 50,000 instances of `bp:Protein`, with results in descending order of number of instances. This user could ask:

```
SELECT DISTINCT ?endpoint ?instances
WHERE { ?dataset v:classPartition [ v:class bp:Protein ; v:distinctSubjects ?instances ] ;
        v:sparqlEndpoint ?endpoint . FILTER(?instances > 50000) }
ORDER BY DESC(?instances)
```

This returns the following result:

?endpoint	?instances
https://www.ebi.ac.uk/rdf/services/reactome/sparql	260,546

As a final example combining scenarios 2 and 3 in the introduction, consider an agent looking for SPARQL endpoints with at least 50 unique images of people, where this agent may ask:

```
SELECT DISTINCT ?endpoint ?imgs
WHERE {
   ?dataset v:classPartition [ v:class f:Person ; v:propertyPartition [
           v:property f:depiction; v:distinctObjects ?imgs ] ] ;
       v:sparqlEndpoint ?endpoint . FILTER(?imgs > 50)
}
ORDER BY DESC(?imgs)
```

This returns the following result:

?endpoint	?imgs
http://eu.dbpedia.org/sparql	4,517
http://eudbpedia.deusto.es/sparql	4,517
http://data.open.ac.uk/query	311
http://apps.morelab.deusto.es/labman/sparql	78

Of course, this is just to briefly highlight three examples of the capabilities of Sportal and the kinds of results it can return. One could imagine various other types of queries that a user could be interested in posing over the Sportal catalogue, which supports a variety of types of queries referring to high-level dataset statistics as well as schema-level information. However, the catalogue does not support finding endpoints mentioning a specific resource or value, nor does it currently support keyword search on the topic of the dataset.

[17] All such answers were generated from the Sportal catalogue in March 2016.

User interface

In order to use the SPARQL interface, the agent must first be familiar with SPARQL, and second must know the IRI of the particular classes and/or properties that they are interested in. To help non-expect users, Sportal also provides an online user interface with a number of functionalities.

First, users can search for specific endpoints by their URL, by the classes in their datasets, and/or by the properties in their datasets. These features are offered by means of auto-completion on keywords, meaning that the agent need not know the specific IRIs they are searching for. Taking a simple example, if a user wishes to find endpoints with instances of drugs, they may type "`drug`" into the search bar and then select one of the presented classes matching that search; once a class is selected, the user is presented with a list of public endpoints mentioning that class, ordered by the distinct subjects for that class partition (as available).

If a user clicks on or searches for an endpoint, they can retrieve all the information available about that endpoint as extracted by the queries previously described, providing an overview of how many triples it contains, how many subjects, how many classes, etc. (as available).

The Sportal user interface also includes some graphical visualisations of some of the high-level features of the catalogue, such as the most popular classes and properties based on the number of endpoints in which they are found, the most common server headers, and so forth. While this may not be of use to a user with a specific search in mind, it offers a useful overview of the content available across all endpoints on the Web, and the schema-level terms that are most often instantiated.

Updates

An important aspect of the Sportal service is to keep up-to-date information about current SPARQL endpoints. Along these lines, we currently recompute the content descriptions every 15 days: we perform a backup of the old catalogue and simply recompute everything from scratch. One shortcoming of this approach is that the catalogue may miss endpoints that were temporarily unavailable during the computation. Currently we do not implement any special workaround for this issue, but we could in future consider importing data from the previous catalogue for endpoints, with a fixed limit for how long into the past we are willing to still consider content descriptions as valid.

Comparison

In Table 11, we compare the Sportal catalogue with two other publicly available services that could be used to find relevant SPARQL endpoints using VoID descriptions: DataHub and VoID Store. Unlike Sportal, both of these services rely on publisher-contributed VoID descriptions.

In the comparison, we include all endpoints that had an associated VoID description in the given service. For DataHub and VoID Store, it is possible to have multiple VoID descriptions associated with an endpoint, and multiple endpoints associated with a VoID file. We count a SPARQL endpoint as available if it could respond with a valid SPARQL response to the query (as used, for example, by the SPARQLES system [10]):

```
SELECT ?s WHERE { ?s ?p ?o } LIMIT 1
```

For DataHub, VoID files are not hosted locally, where links are provided instead. We used the LDspider v1.3 [24] crawler to download the VoID files from these URLs,[18] from which we extract the availability (number of VoID files successfully downloaded) and the content for later statistics.

To give a brief comparison of the coverage of the catalogues, we also display the number of unique classes and unique properties that are associated in each catalogue with at least one endpoint; in more detail, we count the unique classes and unique properties that would be returned for the following queries over the catalogues, respectively:

```
SELECT DISTINCT ?c WHERE { ?s v:class ?c }
```

[18]The exact arguments used were `-s seeds.txt -n -o output.nq -b 0 -any23 -bl .xxx`, indicating to accept all formats supported by any23, to follow redirects but not links (i.e., download seeds), and to not blacklist any file extensions.

Table 11: A comparison of the availability and coverage of SPORTAL, DATAHUB and VOID STORE

Service	Endpoints		Descriptions		Classes	Properties
	Total	*Available*	*Total*	*Available*		
SPORTAL	307	231 (75%)	298	298 (100%)	19,216	46,313
DATAHUB	200	115 (58%)	260	162 (62%)	1,636	829
VOID STORE	118	69 (58%)	148	148 (100%)	30	217

```
SELECT DISTINCT ?p WHERE { ?s v:property ?p }
```

Although this only partially captures the full wealth of information available in VoID, it gives an overview of the diversity of domain terms indexed from endpoints.

From the results, with respect to endpoints, we see that SPORTAL has the broadest coverage: unlike DATAHUB and VOID STORE, it does not require publishers to compute and submit VoID descriptions but rather computes them automatically. For this reason, we see that SPORTAL indexes twice as many available endpoints as DATAHUB and more than three times that of VOID STORE. We also see that the endpoints that SPORTAL indexes have the highest availability ratio: for DATAHUB and VOID STORE, many of the indexed descriptions refer to endpoints that are long dead.

For both SPORTAL and VOID STORE, descriptions are hosted locally, meaning that they are always available when the respective catalogue is available; however, for DATAHUB, 38% of the VoID links provided could not be resolved to RDF content by LDspider.

With respect to the class and property terms mentioned, we see that the SPORTAL catalogue contains orders of magnitude more unique classes and properties than either DATAHUB or VOID STORE.

From these results, we conclude that when compared to DATAHUB and VOID STORE, clients using SPORTAL can expect to find a broader range of relevant endpoints for (e.g.) a broader range of classes and properties, and that the endpoints returned are more likely to be available and to still contain the content in question. Thus we see the benefits of a catalogue based on computing content descriptions rather than relying on those provided by publishers.

Limitations

SPORTAL naturally inherits many of the limitations raised during earlier experiments. For instance, the previous example queries would probably miss endpoints that could not return results for the relevant self-descriptive queries. In general, the catalogue should be considered a best-effort initiative to collect as much metadata about the content of endpoints as possible, rather than a 100% complete catalogue.

Another limitation is that SPORTAL can only help to find endpoints based on the metadata collected from self-describing queries, which mainly centres on the schema terms used. For example, the system cannot help to find endpoints that mention a given literal, or a given subject IRI (which is partially support by VOID STORE using `REGEX` patterns), or to find endpoints based on the text of the description or the tags associated with the relevant dataset (which is supported by DATAHUB), etc.

We must also note that by focusing on the problem of finding relevant SPARQL endpoints, SPORTAL may miss relevant Linked Datasets that do not offer a SPARQL endpoint. According to statistics by Jentzsch et al. [25], only 68% of the Linked Datasets surveyed provided a SPARQL endpoint. Hence, in addition to missing out on endpoints that cannot answer the self-descriptive queries that SPORTAL issues, we also do not cover Linked Datasets without SPARQL endpoints. However, our focus is specifically on the problem of relevant SPARQL endpoints, which we argue is a sufficiently noteworthy problem in and of itself: a problem that merits specialised methods such as those proposed in this paper.

7 Conclusions

In this paper, we proposed a novel cataloguing scheme for helping agents to find public SPARQL endpoints relevant to their needs. Given that the endpoints in question are made available by hundreds of different parties, we chose to investigate a cataloguing system that works with the existing SPARQL infrastructure and, for each endpoint indexed, only requires a working SPARQL interface. We ruled out the option of flooding runtime requests to public SPARQL endpoints looking for the desired content since this would lead to long runtimes and could generate a lot of traffic to public endpoints. Instead, we proposed to use self-descriptive queries to incrementally generate high-level descriptions of the content of public endpoints. We experimented with the performance of running these queries for four datasets and four engines, showing that although Fuseki, Sesame and Virtuoso could successfully answer the queries over small-to-medium-sized datasets, only Virtuoso managed to return results to some queries over larger datasets. We then looked at what sort of success rate public endpoints had in answering these queries, where out of 307 operational endpoints, the ratio of non-empty responses ranged from 25–94% depending on the query in question. Finally we presented details of the SPORTAL prototype that uses the catalogue we have extracted from public endpoints to help users find interesting datasets on the Web.

Although SPORTAL has its limitations, we have shown that it compares favourably with existing services to help clients find SPARQL endpoints: when compared with DATAHUB and VOID STORE, the SPORTAL catalogue has better coverage of available endpoints and, for example, indexes a much broader range of the class and property terms used in the data of remote endpoints. However, it lacks some of the features of these other services: for example, exploring Linked Datasets (and not just SPARQL endpoints) using tags, keyword search over dataset abstracts, searching by resource IRIs, etc.

Our goal in the immediate future is to build upon the existing prototype by seeking feedback from the Linked Data community on what features they feel might be useful, and to gather feedback on the usability of the system. We would also like to investigate fall-back methods of extracting metadata directly from endpoints, such as incremental methods that query, e.g., for statistics about one class/property partition at a time.[19]

The SPORTAL prototype is available online at `http://www.sportalproject.org/`.

Acknowledgements: This publication was supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289, by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004, and by Fondecyt Grant No. 11140900

References

[1] Z. Abedjan, T. Grütze, A. Jentzsch, and F. Naumann. Profiling and mining RDF data with ProLOD++. In *International Conference on Data Engineering (ICDE)*, pages 1198–1201, 2014.

[2] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *International Semantic Web Conference (ISWC)*, pages 18–34. Springer, 2011.

[3] Z. Akar, T. G. Halaç, E. E. Ekinci, and O. Dikenelli. Querying the Web of Interlinked Datasets using VOID Descriptions. In *Linked Data On the Web (LDOW)*. CEUR, 2012.

[4] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing linked datasets. In *Linked Data On the Web (LDOW)*. CEUR, 2009.

[5] S. Auer, J. Demter, M. Martin, and J. Lehmann. LODStats – an extensible framework for high-performance dataset analytics. In *Knowledge Engineering and Knowledge Management (EKAW)*, pages 353–362. Springer, 2012.

[6] C. Basca and A. Bernstein. Querying a messy web of data with Avalanche. *J. Web Sem.*, 26:1–28, 2014.

[7] W. Beek, L. Rietveld, H. R. Bazoobandi, J. Wielemaker, and S. Schlobach. LOD laundromat: A uniform way of publishing other people's dirty data. In *International Semantic Web Conference (ISWC)*, pages 213–228. Springer, 2014.

[19]However, the cost of such an approach would be a prohibitively large number of requests if there are a large number of partitions.

[8] C. Böhm, J. Lorey, and F. Naumann. Creating voiD descriptions for Web-scale data. *J. Web Sem.*, 9(3):339–345, 2011.

[9] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *International Semantic Web Conference (ISWC)*, pages 54–68. Springer, 2002.

[10] C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *International Semantic Web Conference (ISWC)*, pages 277–293. Springer, 2013.

[11] S. Campinas, R. Delbru, and G. Tummarello. Efficiency and precision trade-offs in graph summary algorithms. In *International Database Engineering & Applications Symposium (IDEAS)*, pages 38–47, 2013.

[12] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 23–32, 2002.

[13] R. Cyganiak, H. Stenzhorn, R. Delbru, S. Decker, and G. Tummarello. Semantic Sitemaps: Efficient and Flexible Access to Datasets on the Semantic Web. In *European Semantic Web Conference (ESWC)*, pages 690–704. Springer, 2008.

[14] O. Erling and I. Mikhailov. RDF support in the Virtuoso DBMS. In *Networked Knowledge – Networked Media*. Springer, 2009.

[15] B. Fetahu, S. Dietze, B. P. Nunes, M. A. Casanova, D. Taibi, and W. Nejdl. A Scalable Approach for Efficiently Generating Structured Dataset Topic Profiles. In *European Semantic Web Conference (ESWC)*, pages 519–534. Springer, 2014.

[16] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An Empirical Study of Real-World SPARQL Queries. In *USEWOD*, 2011.

[17] H. Glaser, I. Millard, and A. Jaffri. Rkbexplorer.com: A knowledge driven infrastructure for Linked Data providers. In *European Semantic Web Conference (ESWC)*, pages 797–801. Springer, 2008.

[18] S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In *Scalable Semantic Web Systems Workshop (SWSS)*, 2009.

[19] S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 query language. W3C Recommendation, March 2013.

[20] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *International Conference on World Wide Web (WWW)*, pages 411–420, 2010.

[21] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the Wholst. In *International Semantic Web Conference (ISWC)*, pages 211–224. Springer, 2007.

[22] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool, 2011.

[23] T. Holst and E. Höfig. Investigating the relevance of Linked Open Data Sets with SPARQL queries. In *COMPSAC Workshops*, pages 230–235, 2013.

[24] R. Isele, J. Umbrich, C. Bizer, and A. Harth. LDspider: An open-source crawling framework for the Web of Linked Data. In *International Semantic Web Conference (ISWC) Posters & Demos*. CEUR, 2010.

[25] A. Jentzsch, R. Cyganiak, and C. Bizer. State of the LOD Cloud. Online Report, September 2011. http://lod-cloud.net/state/.

[26] S. Khatchadourian and M. P. Consens. Explod: Summary-based exploration of interlinking and RDF usage in the Linked Open Data Cloud. In *Extended Semantic Web Conference (ESWC)*, pages 272–287. Springer, 2010.

[27] S. Kinsella, U. Bojars, A. Harth, J. G. Breslin, and S. Decker. An interactive map of Semantic Web ontology usage. In *International Conference on Information Visualisation*, pages 179–184, 2008.

[28] A. Langegger and W. Wöß. RDFStats – An Extensible RDF Statistics Generator and Library. In *DEXA Workshops*, pages 79–83, 2009.

[29] T. Lebo, S. Sahoo, and D. McGuinness. PROV-O: The PROV Ontology. W3C Recommendation, April 2013.

[30] J. Lorey. Identifying and determining SPARQL endpoint characteristics. *IJWIS*, 10(3):226–244, 2014.

[31] E. Mäkelä. Aether - generating and viewing extended VoID statistical descriptions of RDF datasets. In *European Semantic Web Conference (ESWC)*, pages 429–433. Springer, 2014.

[32] M. Mehdi, A. Iqbal, A. Hogan, A. Hasnain, Y. Khan, S. Decker, and R. Sahay. Discovering domain-specific public SPARQL endpoints: a life-sciences use-case. In *International Database Engineering & Applications Symposium (IDEAS)*, pages 39–45, 2014.

[33] N. Mihindukulasooriya, M. Poveda-Villalón, R. García-Castro, and A. Gómez-Pérez. Loupe – an online tool for inspecting datasets in the Linked Data cloud. In *International Semantic Web Conference (ISWC) Posters & Demos*. CEUR, 2015.

[34] M. Mountantonakis, C. Allocca, P. Fafalios, N. Minadakis, Y. Marketakis, C. Lantzaki, and Y. Tzitzikas. Extending VoID for expressing connectivity metrics of a semantic warehouse. In *International Workshop on Dataset PROFIling & fEderated Search for Linked Data (PROFILES)*, 2014.

[35] T. Omitola, L. Zuo, C. Gutteridge, I. Millard, H. Glaser, N. Gibbins, and N. Shadbolt. Tracing the provenance of Linked Data using voiD. In *International Conference on Web Intelligence, Mining and Semantics (WIMS)*, page 17, 2011.

[36] H. Paulheim and S. Hertling. Discoverability of SPARQL Endpoints in Linked Open Data. In *International Semantic Web Conference (ISWC) Posters & Demos*, pages 245–248. Springer, 2013.

[37] E. Prud'hommeaux and C. Buil-Aranda. SPARQL 1.1 Federated Query. W3C Recommendation, March 2013.

[38] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *SIGCOMM*, pages 367–378, 2004.

[39] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *European Semantic Web Conference (ESWC)*, pages 524–538. Springer, 2008.

[40] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[41] M. Ripeanu, A. Iamnitchi, and I. T. Foster. Mapping the Gnutella Network. *IEEE Internet Computing*, 6(1):50–57, 2002.

[42] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[43] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: A federation layer for distributed query processing on Linked Open Data. In *Extended Semantic Web Conference (ESWC)*, pages 481–486. Springer, 2011.

[44] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[45] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres. Comparing data summaries for processing live queries over Linked Data. *World Wide Web Journal*, 14(5-6):495–544, 2011.

[46] R. Verborgh, O. Hartig, B. D. Meester, G. Haesendonck, L. D. Vocht, M. V. Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. V. de Walle. Querying datasets on the Web with high availability. In *International Semantic Web Conference (ISWC)*, pages 180–196. Springer, 2014.

[47] G. T. Williams. SPARQL 1.1 Service Description. W3C Recommendation, March 2013.

[48] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

A Prefixes

In Table 12, we list all of the prefixes used in the paper.

Table 12: IRI prefixes used in the paper

Prefix	IRI
bp:	http://www.biopax.org/release/biopax-level3.owl#
dct:	http://purl.org/dc/terms/
dbo:	http://dbpedia.org/ontology/
e:	http://ldf.fi/void-ext#
f:	http://xmlns.com/foaf/0.1/
mo:	http://purl.org/ontology/mo/
p:	http://www.w3.org/ns/prov#
s:	http://vocab.deri.ie/sad#
sp:	http://spinrdf.org/spin#
rs:	http://www.w3.org/2000/01/rdf-schema#
v:	http://rdfs.org/ns/void#
x:	http://www.w3.org/2001/XMLSchema#