SPARQL Web-Querying Infrastructure: Ready for Action?

Carlos Buil-Aranda[1], Aidan Hogan[2],
Jürgen Umbrich[3], and Pierre-Yves Vandenbussche[3]

[1] Department of Computer Science, Pontificia Universidad Católica de Chile
[2] Digital Enterprise Research Institute, National University of Ireland, Galway
[3] Fujitsu (Ireland) Limited, Swords, Co. Dublin, Ireland

Abstract. Hundreds of public SPARQL endpoints have been deployed on the Web, forming a novel decentralised infrastructure for querying billions of structured facts from a variety of sources on a plethora of topics. But is this infrastructure mature enough to support applications? For 427 public SPARQL endpoints registered on the DataHub, we conduct various experiments to test their maturity. Regarding *discoverability*, we find that only one-third of endpoints make descriptive meta-data available, making it difficult to locate or learn about their content and capabilities. Regarding *interoperability*, we find patchy support for established SPARQL features like `ORDER BY` as well as (understandably) for new SPARQL 1.1 features. Regarding *efficiency*, we show that the performance of endpoints for generic queries can vary by up to 3–4 orders of magnitude. Regarding *availability*, based on a 27-month long monitoring experiment, we show that only 32.2% of public endpoints can be expected to have (monthly) "two-nines" uptimes of 99–100%.

1 Introduction

Although there are now tens of billions of facts spanning hundreds of Linked Datasets on the Web, it is still unclear how applications can begin to make effective use of these data. A foundational requirement for any application is the ability to discover, access and query the data. Addressing this need, SPARQL—*the* query language for RDF—was first standardised in 2008 [18], and an extension in the form of SPARQL 1.1 was also recently standardised [12]. SPARQL has thus been a core focus of research and development for Semantic Web technologies in the past five years, with various research proposals, benchmarks, open-source and commercial tools emerging to address the challenges of processing SPARQL queries efficiently, at large scale and in distributed environments.

These advances in SPARQL technology and tools have been paralleled by the deployment of public SPARQL endpoints on the Web: to date, over four hundred such endpoints have been announced on the DataHub site[4], with approx. 68% of official "LOD Cloud" datasets claiming to host an endpoint.[5] Prominent

[4] A Linked Data catalogue: http://datahub.io/group/lod (l.a.: 2013-05-10)
[5] http://lod-cloud.net/state/ (l.a.: 2013-05-10)

endpoints are now logging significant levels of traffic, where, e.g., studies of logs from the DBpedia SPARQL engine reveal that it is servicing in the order of hundreds of thousands of queries per day [17,8]. Answering queries that span the content of these endpoints is then an open research question, and one that has been tackled by a host of works on SPARQL federation [12,1,3,19,20,16,5,11].

Despite all of this momentum, however, few applications are emerging to exploit this novel querying infrastructure. In this paper, we thus ask: IS THIS NOVEL DECENTRALISED SPARQL INFRASTRUCTURE READY FOR ACTION? Focusing on the technical challenges (where we rather refer to, e.g., [13, § 2], for content-related discussion), we take a list of 427 public SPARQL endpoints from the DataHub site[6] and present the following core experiments:

§ 2 We first look at the DISCOVERABILITY of the 427 endpoints, analysing how endpoints can be located, what meta-data are available for them, etc.

§ 3 We analyse INTEROPERABILITY, using SPARQL 1.0 and SPARQL 1.1 test-case suites to identify features (not) supported by these endpoints.

§ 4 We tackle EFFICIENCY by testing the time taken by individual endpoints to answer generic, content-agnostic SPARQL queries over HTTP.

§ 5 We measure RELIABILITY based on a 27-month long monitoring experiment of the uptimes of public SPARQL endpoints.

EXPERIMENTAL OVERVIEW: All results were collated in May 2013. The most recent list of endpoints that we retrieved from DataHub contained 427 SPARQL endpoint URLs. Here considering "pay-level-domains"—the level at which a domain can be registered and must be individually paid for—we found 159 domains hosting the 427 endpoints: a mean of 2.7 ± 6.3 endpoints per domain. 54 (12.6%) endpoints are hosted by `rkbexplorer.com`, 37 (8.7%) by `bio2rdf.org`, 36 (8.4%) by `talis.com`, 24 (5.6%) by `eagle-i.net`, 20 (4.7%) by `kasabi.com`, etc.

For running queries, we use Apache Jena ARQ 2.9.3 requesting XML or RDF/XML results, with a first-result timeout of 1 minute and an overall timeout of 15 minutes. We run queries sequentially and enforce a politeness wait of one second between the end of execution of one query and the start of the next.

For reproducibility, all code, queries and results relating to this paper—including larger versions of the graphical figures presented herein—are available online at http://labs.mondeca.com/sparqlEndpointsStatus/iswc2013/.

EXAMPLE USE-CASE: To ground our discussion, we refer throughout the paper to a hypothetical use-case involving a plug-in for online video sites such as YouTube, Vimeo, etc. The plug-in detects protagonists of the video (e.g., the TV series that the clip is from, the music artist for the track, the location where the video is taken, etc.) and attempts to discover and query relevant public SPARQL endpoints for meta-data about the protagonists (and the relationships between them) such that can be processed and presented to the user in a side-bar.

[6] The raw list is fetched from the DataHub API: http://datahub.io/api/2/search/resource?format=api/sparql&all_fields=1&limit=10000 (l.a.: 2013-05-10)

2 Endpoint Descriptions

A prospective consumer of SPARQL endpoints first needs to DISCOVER relevant endpoints that support the features they require for their application domain. We identify two main vocabularies that current SPARQL endpoints use to describe their features and content: VoID [2] and SPARQL 1.1 Service Descriptions [21].

2.1 VoID Catalogues

A consumer may (like us) use the DataHub catalogue to collect endpoint URLs. However, to find *relevant* endpoints, the consumer will need descriptions of their content. Relatedly, the VoID vocabulary can be used to describe an RDF dataset, including statistics about size, schema terms used, frequencies of terms, access mechanisms, URI patterns mentioned, a link to an OpenSearch Description Document, as well as a link to the endpoint in question.

EXPERIMENTS: We identify two primary online catalogues that an agent could query to find relevant SPARQL endpoints. The first is the aforementioned DataHub catalogue. The second is the "VoID store"[7] hosted by the RKBExplorer project [9]. We issue the following template query to both:

```
PREFIX void: <http://rdfs.org/ns/void#>
SELECT DISTINCT ?ds
WHERE { ?ds a void:Dataset ; void:sparqlEndpoint %%ep . }
```

We instantiate this query for each of the 427 endpoints by substituting the placeholder "`%%ep`" with the given service URI. We then execute each query against the two catalogue interfaces. We also execute the query directly against the endpoint itself in case it indexes its own VoID description and look in the `http://{domain}/.well-known/void` location recommended for use with VoID.[8]

RESULTS: The DataHub catalogue returned results for 142 endpoints (33.3%) and the VoID store for 96 endpoints (22.4%). Many of these VoID URLs referred to the root domain folder or a `models/` folder with filename `void.ttl`. We found that only 69 endpoints (16.2%) indexed VoID data about themselves. Looking up the `.well-known` location yielded no VoID files. In summary, we see that discoverable VoID descriptions for the content of public endpoints are sparse.

2.2 SPARQL 1.1 Service Descriptions

Once the consumer has found an endpoint relevant to their needs (be it using a VoID description or other means), they will need meta-data about the *capabilities* of the endpoint: which query features, I/O formats or entailments are supported; how default and named graphs are configured; etc. Such capabilities can be described using the SPARQL 1.1 Service Description (SD) vocabulary [21].

[7] http://void.rkbexplorer.com/sparql/ (l.a.: 2013-05-10)
[8] http://vocab.deri.ie/void/autodiscovery; (l.a.: 2013-05-10).

Table 1. Number of endpoint descriptions containing SD and VoID properties

Predicate	№	Predicate	№
sd:resultFormat	38	void:sparqlEndpoint	5
sd:feature	36	void:classes	2
sd:supportedLanguage	36	void:datadump	2
sd:url	34	void:triples	2
sd:endpoint	33	void:vocabulary	2

Table 2. Server names in `HTTP Get` responses

Server-field Prefix	№
Apache	157
Virtuoso	71
Jetty	25
nginx	23
Fuseki	6
4s-httpd	3

Experiments: The Service Description of an endpoint can be retrieved by dereferencing the endpoint URI itself [21]. We thus performed a `HTTP Get` request for each of the 427 endpoint URIs, following redirects, requesting RDF formats (viz. RDF/XML, N-Triples, Turtle or RDFa). We also check the resulting HTTP headers for interesting meta-data relating to the endpoint.

Results: In total, 151 lookups (35.4%) returned a `200 OK` response code. Only 51 endpoints (11.9%) returned an RDF-specific content-type (RDF/XML in all cases) and 95 endpoints (22.2%) returned the typical HTML query interface (without embedded RDFa). We received 173 "`4xx`" responses (40.5%) indicating client-side errors and 47 "`5xx`" responses (11%) indicating server-side errors. (Section 5 will refer to these availability issues in greater detail.)

We then inspected the dereferenced content for SD and VoID meta-data using Apache Any23 Java library[9] to extract RDF. Table 1 lists the top-5 SD and VoID properties by the number of descriptions they appear in (note: `sd:url` is a non-standard term). We found 39 endpoints (9.1%) offering some SD meta-data and a handful providing VoID meta-data by dereferencing.

Finally, we checked the HTTP response headers for meta-data about the underlying SPARQL service. The most interesting meta-data came from the `Server` field, which sometimes identified the SPARQL engine powering the endpoint. Table 2 lists some common values found: though most values still referred to generic Web servers/proxies such as Apache, we could identify *Virtuoso*, *Fuseki* and *4store* SPARQL implementations from the `Server` field.

2.3 Summary of Discoverability

We identify two existing RDF vocabularies that can be used to "advertise" the content and features of an endpoint respectively, potentially allowing for remote discovery. However, locating such descriptions is difficult. Catalogues provide VoID descriptions for about one-third of the endpoints, and other methods of finding VoID descriptions are largely unsuccessful: in most cases, the VoID descriptions (used, e.g., for federation [11]) probably do not exist. Where available, SD meta-data is easy to find using "follow-your-nose" principles, but being a relatively new W3C proposal, is supported by fewer than one-tenth of endpoints. Furthermore, it is unclear if VoID and SD alone are enough to enable mature

[9] http://any23.apache.org/ (l.a.: 2013-05-10)

auto-discovery of endpoints; however, adding new vocabulary is rather straightforward once the mechanisms for discovering such descriptions are in place.

Example Use-case: The use-case application first needs to find SPARQL endpoints relevant to the various types of protagonists detectable in, e.g., YouTube videos. Although there are a number of potentially relevant endpoints on the Web (e.g., BBC Music, BBC Programmes, DBTune, DBpedia, FactForge, Linked Movie DataBase, MusicBrainz, notube, etc.) these are difficult to discover automatically, and would probably require manually going through endpoint catalogues to find. Furthermore, once these relevant endpoints are identified, information about their indexed content (coverage, topic, vocabularies, etc.), functionalities (full-text search, entailment, etc.) and policies (result limits, timeouts, max query rate, etc.) is not available. The developers of the plug-in will be hampered by a lack of appropriate meta-data for individual endpoints.

3 Features Supported

Service Descriptions are still scarce, making it difficult to know the functionalities of an endpoint. Furthermore, endpoints may be non-compliant for the features they claim to support. In this section, we thus empirically analyse the SPARQL and SPARQL 1.1 features supported by the 427 public endpoints.

3.1 SPARQL 1.0 Standard Features

Experiments: We first analyse which core SPARQL 1.0 features the servers support. For this, we use a subset of the Data Access Working Group test-cases for SPARQL 1.0,[10] which tests features that a compliant SPARQL implementation must fulfil. We omit syntax tests and focus on core functionalities.[11] For each query, we test if it runs without throwing an exception: without control over content, we cannot validate results and hence we may overestimate compliance. Conversely, although features may be supported, queries can throw exceptions due to, e.g., timeouts; here we may underestimate feature support.

When presenting the compliance for individual queries, we use an abbreviated feature algebra (cf. Figure 1). We first evaluate support for SPARQL SELECT queries with a single-triple pattern (sel[.]) and use of core query-clause operators: joins (sel[join]), OPTIONAL (sel[opt]) and UNION (sel[union]). We also include a query returning 0 results (sel[empty]). Next, we evaluate compliance for FILTER (fil(.)). For that, we use several official SPARQL operators like regex, IRI and blank node checks (labelled intuitively). We also check support for datatypes (numeric, strings and booleans). Finally for filters, we evaluate the bound-checking function. We then evaluate dataset definition support, checking compliance of FROM (NAMED) and GRAPH operators in

[10] http://www.w3.org/2001/sw/DataAccess/tests/r2 (l.a.: 2013-05-10)
[11] Queries available at http://labs.mondeca.com/sparqlEndpointsStatus/iswc2013/.

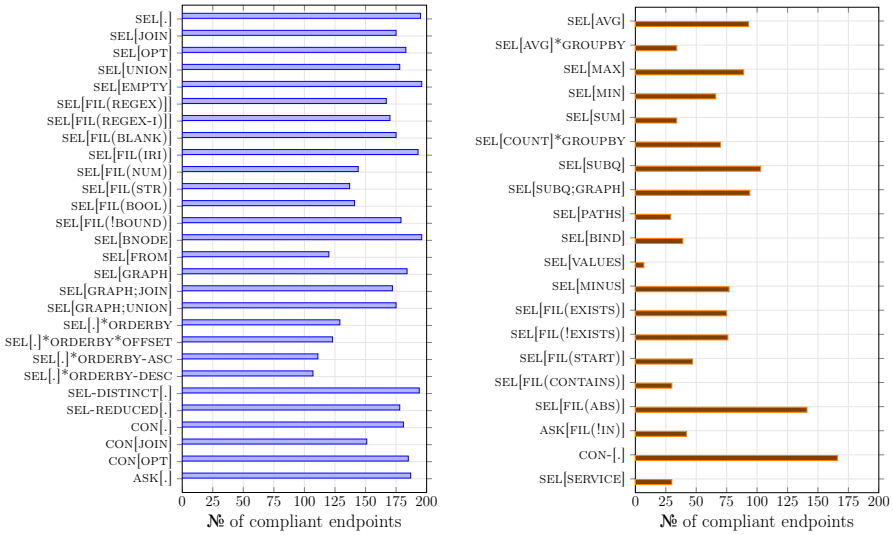

Fig. 1. SPARQL compliance results (SPARQL 1.0 left, SPARQL 1.1 right)

combination with other standard features. We next check solution modifiers: ORDER BY, LIMIT and OFFSET (DESC|ASC), as well as the DISTINCT and REDUCED keywords for select queries. Finally, we check support for CONSTRUCT and ASK query-types (we omit DESCRIBE since support is optional).

Results: The results for SPARQL compliance are presented in Figure 1 (alongside SPARQL 1.1 results discussed later). More than half of the 427 endpoints threw exceptions for all queries. The most common types of exception were connection errors and HTTP-level issues (e.g., `403`s and `404`s), corresponding with previous observations when dereferencing the endpoint URI.

Other exceptions depended on the query being sent. Features such as ORDER BY, FROM and data-type filters were particularly problematic. For example, while SEL[.] was answered by 195 endpoints without exception, adding the ORDERBY feature meant that the queries were only answered by 129 endpoints. Some of the differential errors were, for example, content type issues ("`Endpoint returned Content-Type: text/xml which is not currently supported for SELECT queries`"; running these queries manually against the endpoints gave different generic errors). Non-standard error-reporting made identification of the root problem difficult (for us and for potential consumers). For example, a prevalent issue was timeouts (since some queries require long processing times) but only 5 endpoints clearly indicated when a query was too expensive.

3.2 SPARQL 1.1 Early Adopters

SPARQL 1.1 has been recently standardised and adds many new features, including sub-queries, aggregates, federation, entailment, updates and so forth. We now see how many endpoints support novel SPARQL 1.1 features.

Experiments: We now select tests from the SPARQL-WG for SPARQL 1.1.[12] We omit testing for SPARQL 1.1 Update since we (hopefully) will not have write privileges for public endpoints. We do not test entailment since, without knowledge of the content, we cannot verify if results are entailed or not.

We first test support for aggregates, where expressions such as average, maximum, minimum, sum and count can be applied over groups of solutions (possibly using an explicit GROUP BY clause). We then test support for sub-queries (SUBQ) in combination with other features. Next we test support for property-paths (PATHS), binding of individual variables (BIND), and support for binding tuples of variables (VALUES). We also check support for new filter features that check for the existence of some data (MINUS, EXISTS), and some new operator expressions (STARTS and CONTAINS for strings; ABS for numerics). Finally, the last three queries test a miscellany of features including NOT IN used to check a variable binding against a list of filtered values (!IN), an abbreviated version of CONSTRUCT queries whereby the WHERE clause can be omitted (CONS-), and support for the SPARQL SERVICE keyword.

Results: In contrast to core SPARQL queries, we now receive syntax errors that indicate when the SPARQL 1.1 feature in question is not supported. However, we also encountered timeout errors where the feature may be supported but could not return a valid response. On the right hand side of Figure 1, we see that support for SPARQL 1.1 features is more patchy than for core SPARQL. The abbreviated CONSTRUCT syntax was (relatively) well supported. Conversely, certain queries were executed by fewer than 40 endpoints, including features relating to string manipulation like CONTAINS, aggregates like SUM, property paths and VALUES. The differing amount of errors in aggregate functions like SUM or MIN versus AVG or MAX are due to timeouts or errors when manipulating incorrect datatypes. We highlight support of the SERVICE keyword in 30 SPARQL endpoints, which can be used to integrate results from several SPARQL endpoints (tested by invoking DBpedia's SPARQL endpoint).

3.3 Summary of Interoperability

We have seen that over half of the endpoints were not available to answer queries during these experiments; many endpoints are permanently down, as we will see later in our availability study. Otherwise, of the core SPARQL features, simple queries involving SELECT and ASK were executed broadly without exceptions. However, many endpoints threw exceptions for more complex queries, particularly ORDER BY. We also found early adoption of some SPARQL 1.1 features amongst the endpoints (and it is indeed still early days).

For the purposes of interoperability, consumers should be able to expect broad compliance with the original SPARQL specification (and SPARQL 1.1 in future). We also highlight that standardisation of error messages from endpoints would greatly aid interoperability. On a side note, we draw attention to the lack

[12] http://www.w3.org/2009/sparql/docs/tests/data-sparql11/ (l.a.: 2013-05-10)

of full-text search functionality in the SPARQL standard. Such a feature is commonly required by users of SPARQL stores and thus, many different SPARQL implementations support custom functions for full-text search.[13] The heterogeneity for syntax and semantics of full-text search across different engines—and thus across different endpoints—hinders interoperability for consumers requiring this commonly implemented feature, where even ad hoc standards would be welcome.

Example Use-case: Per the results of Figure 1, certain query templates used by the plug-in will succeed for some endpoints and fail for others. For example, some endpoints may fail to order the album release dates of a given artist chronologically due to problems with the ORDERBY feature. Other endpoints may fail to count the number of movies in which two actors co-starred due to a lack of support for the SPARQL 1.1. GROUPBY and COUNT features (of course, SPARQL 1.1 was only recently standardised). Finally, the lack of full-text search makes finding resource descriptions for protagonists difficult.

4 Performance

We now look at a number of performance issues. We first look at the rate at which endpoints can stream SPARQL results over HTTP, and as a side result, we capture the result-size thresholds enforced by many endpoints. Next we look at how fast different types of atomic lookups can be run over the endpoints. Finally, we measure generic join performance. Throughout, we run queries twice to compare cold-cache (first run) and warm-cache (second run) timings.

4.1 Result Streaming

Experiments: We first measure the performance of the endpoints for streaming a large set of results using a query that should be inexpensive to compute:

```
SELECT * WHERE { ?s ?p ?o } LIMIT 100002
```

We add an additional 2 results to distinguish cases where the endpoint implements a results-size threshold of 100,000.

Results: Overall, 233 endpoints (54.6%) returned no results. Only 57 endpoints (13.3%) returned 100,002 results. We found 68 endpoints (15.9%) returning a "round number" of results ($x = y \times 10^z$ for $y, z \in \mathbb{N}$ and $1 \leq y \leq 15$) suggestive of a fixed result-size threshold; Table 3 lists the thresholds we encountered up to 100,000. The remaining 69 endpoints (16.2%) returned a non-round number of results less than 100,002: almost all returned the same number of results in cold-cache and warm-cache runs suggesting that the endpoints index fewer than 100,002 triples. However, 10 endpoints from `rkbexplorer.com` returned 99,882–99,948 triples, indicating a possible bug in their `LIMIT` counting.

[13] These include: `onto:luceneQuery` (OWLIM), `fti:match` (AllegroGraph), `bif:contains` (Virtuoso), `pf:textMatch` (Jena), `text:dmetaphone` (4store), `search:text` (Sesame).

Table 3. Result-size thresholds

№ of Results	№ of Endpoints
500	1
1,000	3
1,500	1
5,000	1
10,000	49
20,000	2
40,000	1
50,000	3
100,000	7
TOTAL:	**68**

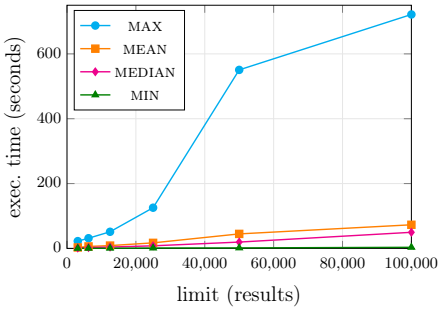

Fig. 2. Comparing different limit sizes

Regarding runtimes, we only consider endpoints that returned at least 99% of the limit quota (99,000 results), leaving 75 endpoints (17.6%). The mean cold-cache query time for these endpoints was 72.3 s (±97.7 s), with a median of 50.7 s, a minimum of 5.5 s and a maximum of 723.8 s. The high standard deviation indicates a significant positive skew in the performance of endpoints. The warm cache values were analogous with a mean of 72.7 s (±97.7 s); the differences with cold cache times were not significant ($p \approx 0.966$ using the Wilcoxon Signed Rank test: a non-parametric test for non-normal matched samples). To estimate the overhead of HTTP connections for the queries, we also varied the `LIMIT` sizes for 50,000, 25,000, 12,500, 6,250 and 3,125 (i.e., $\frac{100{,}000}{2^n}$ for n from 0–5), again considering 75 endpoints that filled 99% of all quotas. Figure 2 presents the max., mean, median and min. query times for varying `LIMIT` sizes. We see that the max. outlier is not due to a fixed HTTP cost. Taking the mean results, successively doubling the `LIMIT` from 3,125 five times lead to time increases of 2×, 1.4×, 2×, 2.6× & 1.6× respectively. Though means are affected by outliers, query times are not dominated by fixed HTTP costs.

4.2 Atomic Lookups

EXPERIMENTS: Next we look at the atomic lookup performance of the SPARQL endpoints. We pose each endpoint with a single lookup as follows:

```
ASK WHERE { ?s <y> <z> }
```

We use the URI abbreviations `<x>`, `<y>` and `<z>` to denote an arbitrary fresh URI not used elsewhere and that will not be mentioned in the content of the endpoint. We use fresh URIs so as to create lookups that will not return results from the endpoint, thus factoring out the content stored by that endpoint while still executing a lookup. We use seven types of single-pattern `ASK` queries of the above form: calling the above query an $\texttt{ASK}_{po}$ query since the predicate and object are constant, we also generate $\texttt{ASK}_{s}$, $\texttt{ASK}_{p}$, $\texttt{ASK}_{o}$, $\texttt{ASK}_{sp}$, $\texttt{ASK}_{so}$, $\texttt{ASK}_{po}$, and $\texttt{ASK}_{spo}$ queries using fresh URIs each time.

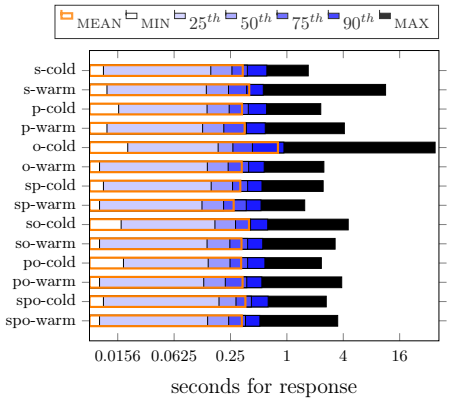

Fig. 3. Runtimes for ASK queries (%iles)

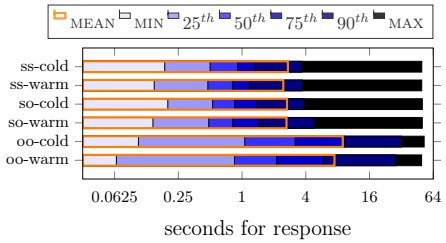

Fig. 4. Runtimes for join queries (%iles)

Results: We consider only 139 comparable endpoints (32.6%) that successfully returned `false` in all runs.[14] The query-time results for these endpoints are compared in Figure 3. In the mean case, such lookups take between 270–400 ms, with the exception of $\texttt{ASK}_o$, which took a mean of 800 ms in the cold run due to an outlier. The MAX and 95^{th}-%ILE rows indicate a small number of endpoints taking several seconds even for such simple queries. With respect to warm vs. cold cache times for the same queries, although the mean warm cache times are sometimes slower due to outliers, the median warm cache times are slightly faster in all seven cases, with significant results ($p < 0.05$, again using the Wilcoxon Signed Rank test) for all but the $\texttt{ASK}_{po}$ case (for which $p \approx 0.108$).

4.3 Joins

Experiments: We now look at join performance. In order to minimise the content of the endpoints as a factor, we again rely on queries that perform lookups on fresh URIs not appearing in the endpoint's content:

```
SELECT DISTINCT ?s ?p2
WHERE { ?s ?p ?o .
OPTIONAL {?s ?p2 <x> .} }
LIMIT 1000
```

```
SELECT DISTINCT ?s ?p2
WHERE { ?s ?p ?o .
OPTIONAL {<x> ?p2 ?s .} }
LIMIT 1000
```

```
SELECT DISTINCT ?o ?p2
WHERE { ?s ?p ?o .
OPTIONAL {<x> ?p2 ?o .} }
LIMIT 1000
```

These three queries respectively measure the performance of three types of joins: subject–subject joins (s–s), subject–object joins (s–o) and object–object joins (o–o). Each query asks to perform a lookup for 1,000 unique terms: the OPTIONAL clause ensures that the query stops after 1,000 terms are returned while allowing to perform lookups with a controlled number of results (in this

[14] 53 of the 54 endpoints on the `rkbexplorer.com` site failed for $\texttt{ASK}_{spo}$. Furthermore, the endpoint at http://sparql.data.southampton.ac.uk/ returns `true` for any ASK query requesting XML results. Similarly, http://dbtune.org/classical/sparql/ often erroneously returns `true`, esp. for warm-cache queries. (l.a.: 2013-05-10)

case zero). Returning the ?p2 variable ensures that the OPTIONAL clause cannot be "compiled away" since the SPARQL engine must execute the clause for each unique term to check whether ?p2 has any bindings (or is unbound).

Results: Figure 4 presents join-query performance for 122 endpoints (28.6%) that returned at least 990 results for all six runs. Warm cache queries are faster for the same query type, with significant results (using Wilcoxon Signed Rank tests) for s–s joins ($p \approx 0.009$) and o–o joins ($p \approx 0$), but not for s–o joins ($p \approx 0.118$). In the mean case, the s–* joins took around 2.5 s whereas the o–o joins took 9 s cold cache and 7.5 s warm cache; slower o–o joins are also evident in the median case and are not due to outliers ($p \approx 0$). We suspect that object terms (e.g., classes) may be mentioned in thousands of triples and it may thus require streaming many tuples to meet the 1,000 unique object-term quota.

4.4 Summary of Performance

First, we see that the reliability of many endpoints becomes worse than previously encountered when issued with a series of non-trivial queries: despite asking queries that would generally take less than a few seconds to process, and despite waiting one second between queries, we saw variable reliability of endpoints across multiple runs. Second, only 57 endpoints returned 100,002 results when requested, where we found at least 68 endpoints implementing result-size thresholds (such a feature should ideally be noted in an endpoint's Service Description). Third, we find small but often significant improvements in warm cache performance. Fourth, median performance times were generally quite reasonable: 0.5 ms per streaming result, 300 ms per ASK query, and 1 ms per join result; we can thus estimate an average fixed HTTP cost of ∼300 ms per query.

As a key overall conclusion, there is high variance (i.e., positive skew) in performance for different endpoints when executing analogous queries. To illustrate this point, in Figure 5 we present a Lorenz curve for the three types of performance results. For cold-cache runs of each experiment, we apply the same filtering of empty or partial results as before and sum up the total execution time. We then sort endpoints in ascending order of the amount of execution time taken for the experiment (summating variations of queries for ask and join) and plot the Lorenz curve based on x ratio of the fastest endpoints taking y ratio of the total time taken. For example, taking $x = 0.5$, we can say that during the join experiment, 50% of the fastest endpoints took up about 10% of the experiment time, with the slower 50% taking the remaining 90%. We can also say that the slowest 10% of endpoints took 45% of the execution time. The equality line plots the Lorenz curve assuming equally performant endpoints ($x = y$). Notably, the Lorenz curves show a similar (skewed) characteristic across all three experiments with increasing positive skew for increasingly complex queries.

Example Use-case: Performance issues will have obvious consequences for our use-case plug-in. Even for atomic ASK queries, some endpoints take multiple seconds to respond. Given that the performance of different endpoints over HTTP can vary by orders of magnitude, the users of the plug-in may have to

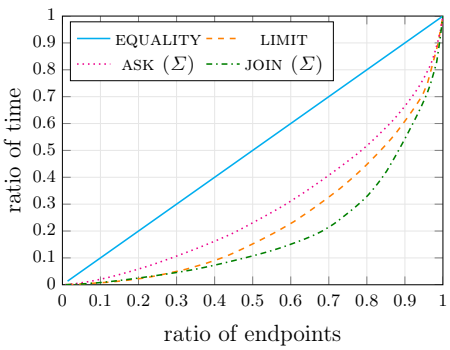

Fig. 5. Lorenz curves for total execution times of streaming (limit), lookup (ask) and join queries.

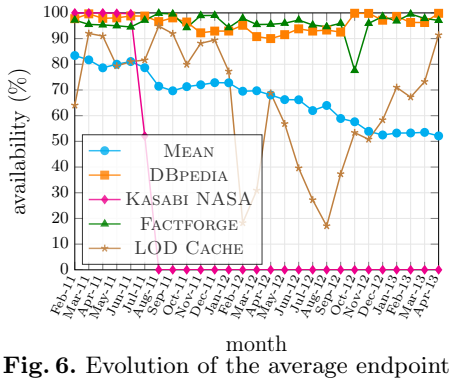

Fig. 6. Evolution of the average endpoint availability between February 2011 and April 2013 (inclusive)

wait while results for the video are being collected from slow endpoints, creating bottle-necks that affect the quality and responsiveness of the application. Furthermore, the application developers will have to account for the possibility that the results they receive from the endpoint may have had a threshold (silently) applied, particularly if more than 10,000 results are requested.

5 Availability

The previous sections repeatedly suggest that SPARQL endpoint availability poses a huge obstacle that applications relying on SPARQL technology have to face. We now explore this issue, considering an endpoint available if it is (1) accessible using the HTTP SPARQL protocol [6]; (2) able to process a SPARQL compliant query [18]; *and* (3) able to respond using SPARQL formats. Given a set of requests issued at fixed time intervals over a given time period, the availability of an endpoint in that period is defined as the ratio of the total requests that succeed vs. the total number of requests made.

5.1 Status monitoring

Experiment: To accurately assess the availability of public SPARQL endpoints, we have been continuously monitoring all SPARQL endpoints listed in DataHub on an hourly basis since February 2011. As of the end of April 2013, when we collated our results, more than seven million test queries had been executed. The evolving availability statistics for endpoints are published to a live, online service.[15] In order to accommodate patchy SPARQL compliance (cf. Section 3), we try two queries to test availability for each endpoint:

[15] SPARQL Endpoint Status: http://labs.mondeca.com/sparqlEndpointsStatus/index.html (l.a.: 2013-05-10)

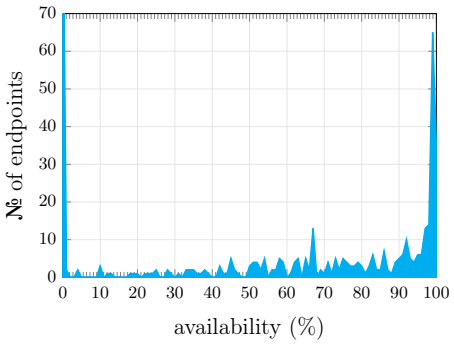

Fig. 7. Endpoint availabilities averaged from February 2011 to April 2013

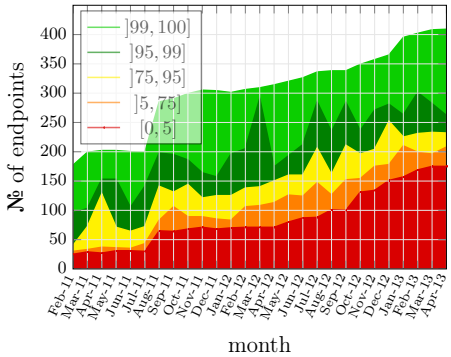

Fig. 8. Evolution of endpoint availabilities from February 2011 to April 2013

```
ASK WHERE { ?s ?p ?o . }
```

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

If the ASK query fails (e.g., is not supported) we try the SELECT query. We consider the request successful if we get a valid response for either query.

Results: Based on this 27-month-long experiment, Figure 6 shows that the mean endpoint availability has been decreasing over time (from 83% in February 2011 to 51% in April 2013), explaining endpoint errors noticed in previous sections. However, mean availability is affected by a growing number of offline endpoints, exemplified by Kasabi NASA in Figure 6. The mean trend is not followed by all endpoints: for instance, the availability of the DBpedia endpoint[16] has always been above 90%, with a mean of 96%. We also highlight the variable availability profile of two prominent "centralised" warehouses—LOD Cache and FactForge—that index third-party data.

The variance of endpoint-availability profiles is illustrated by the distribution presented in Figure 7, where many endpoints fall into one of two extremes: 24.3% of endpoints are always down whereas 31% of endpoints have an availability rate higher than 95%. In Figure 8, we additionally plot the evolution of these distributions across the 27 month long lifespan of the experiment. We again see many endpoints falling into one of two extremes, with few endpoints in the aggregate $[5, 95]$ interval. In addition, we see a continuous increase in the number of new endpoints being listed on DataHub over the past months while, conversely, the number of endpoints going offline continues to grow.

5.2 Summary of Availability

We identify 5 endpoint availability intervals of interest (cf. Figures 7 & 8):

0% to 5% This category, covering 29.3% of endpoints on average, may be qualified as "unreliable". No application should rely on such endpoints.

[16] See http://dbpedia.org/sparql (l.a.: 2013-05-10)

5% to 75% This "low reliability" category represents 8.7% of endpoints on average. This category could be used by applications for non-critical updates that can deal gracefully with no response from endpoints.

75% to 95% On average, 12.4% of endpoints fall into this "medium reliability" interval, usable for applications with intermittent update requirements.

95% to 99% This "high reliability" category covers 17.4% of endpoints. Applications that require real-time responses could target this interval (or higher).

99% to 100% On average, 32.2% of endpoints belong to this "very high reliability" interval. Applications with strong reliability requirements should exclusively use this category to deliver good quality of service.

The apparent overall decline in endpoint availability is possibly an effect of maturation. SPARQL is currently moving away from experimentation [14], leaving permanently offline endpoints in its wake (e.g. Kasabi endpoints) with fewer new "experimental" endpoints being reported. However, other endpoints (e.g., data.gov) are supported by well-established stakeholders (e.g., U.S. Government) and are part of a sustainable policy to deliver a high quality of service to end-user applications. Hence we see the large division in availability profiles where those with low availability will inevitably die off and where those with higher availability (hopefully) tend towards maturation. Informally, in the nomenclature of Gartner's hype cycle, we can conjecture that SPARQL has now gone past the "Peak of Inflated Expectations" leaving some dead endpoints behind.

Example Use-case: With endpoint availability being as patchy as illustrated in Figure 8, the developers of our use-case plug-in cannot rely on individual endpoints to serve the content that users require when users require it. Only about one-third of the endpoints fall into the very-high reliability bracket, and even still, an availability of 99% may not be enough, especially when considering that the plug-in may rely on multiple such endpoints at any given time: if the developers wish to rely on a federation of multiple independent endpoints, they must further consider that the availability of the federation as a whole will effectively be a product of the individual availabilities. Even more worryingly for the developers, endpoints sometimes disappear permanently. For example, the BBC Music, BBC Programmes and MusicBrainz endpoints are now permanently offline: the hosting company (Talis) has discontinued these services. If the plug-in were dependent on the content provided by these services, their sudden discontinuation would have severe impact on the use-case application.

6 Conclusion

More than five years on from the original SPARQL standard, we have analysed four key issues in terms of the maturity of the current SPARQL infrastructure available on the Web today: DISCOVERABILITY, INTEROPERABILITY, EFFICIENCY and RELIABILITY. Our experiments have undoubtedly painted a mixed picture of the maturity of the SPARQL infrastructure: applications built on top of this infrastructure must cope with the intrinsic characteristic of imperfection and

varying degrees of reliability that one often finds on the Web. Our experiments are designed to help inform (potential) practitioners with respect to the potential pitfalls and opportunities offered by this fledgling (and imperfect) infrastructure.

With respect to disseminating results, we continue to host the "SPARQL Endpoint Status Website"[15] for the reference of practitioners such that they can identify the profile of availability that endpoints of interest fall into. In the near future, we hope to offer a more complete monitoring tool that runs our experiments at regular intervals, which will help potential consumers identify discoverable, interoperable, efficient and reliable SPARQL endpoints on the Web. We are also looking into making these up-to-date results available as RDF through SPARQL, allowing clients to query the behaviour of monitored endpoints.

As for the initial question raised at the outset of this paper: IS THIS NOVEL DECENTRALISED SPARQL INFRASTRUCTURE READY FOR ACTION? Certainly for the types of applications exemplified by our highlighted use-case—and even aside from those experimental endpoints that skew our results in a negative direction—the most appropriate answer for the moment would seem to be: *not yet*. However, we hope that this paper will highlight the current challenges faced and suggest some open issues for the community to address:

Discoverability: Aside from VoID and SPARQL Service Descriptions, how should SPARQL endpoints describe their content? How can these descriptions be advertised and discovered by potential clients? Perhaps works on structural summaries [10], cataloguing in dataspaces [7] or routing indexes in P2P systems [15], may yield insights on how best to describe, advertise and automatically discover and invoke public SPARQL endpoints.

Interoperability: Certain endpoints do not support certain SPARQL features, or implement hidden result limits or query timeouts, etc. Thus, how can Quality-of-Service guarantees and other API policies be made explicit for clients? Furthermore, how can error reporting be standardised? Such features would allow clients to implement remote exception handling (e.g., if a query times out, ask a simpler query). De facto standards for features like full-text search would also be beneficial for clients.

Performance: Further work is perhaps required on optimising local evaluation of SPARQL queries. Efficient support for new SPARQL 1.1 features such as property-paths, aggregates and entailment is still an open question. An alternative direction is to define lightweight subsets of SPARQL for which queries can be executed efficiently and accurate cost models developed, allowing endpoints to make more reliable guarantees.

Availability: Remote server down-times can be mitigated by local caching, mirroring and other replication techniques, where database caching techniques (e.g., DBCache [4]) seem relevant. Broadly speaking, works on partition tolerance or overlay networks in distributed systems may also yield insights into the question of availability and fault-tolerance.

In conclusion—and as this paper has shown—there are still many challenges that must be addressed before SPARQL infrastructure will be ready for action.

Acknowledgements: *This work was supported by Fujitsu (Ireland) Ltd. & by Science Foundation Ireland under Grant № SFI/08/CE/I1380 (Lion-2). Carlos Buil-Aranda was supported by CONICYT/FONDECYT project № 3130617.*

References

1. M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An adaptive query processing engine for SPARQL endpoints. In *ISWC*, pages 18–34. Springer, 2011.
2. K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing linked datasets. In *LDOW*. CEUR (Vol. 538), 2009.
3. C. Basca and A. Bernstein. Avalanche: Putting the spirit of the web back into semantic web querying. In *SSWS*, pages 524–538. CEUR (Vol. 669), 2010.
4. C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with dbcache. *IEEE Data Eng. Bull.*, 27(2):11–18, 2004.
5. C. Buil-Aranda, M. Arenas, O. Corcho, and A. Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *JWS*, 18(1):1–17, Jan. 2013.
6. K. G. Clark, L. Feigenbaum, and E. Torres. SPARQL Protocol for RDF. W3C Recommendation, January 2008.
7. M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.
8. M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. D. L. Fuente. An empirical study of real-world SPARQL queries. In *USEWOD Workshop*, 2012.
9. H. Glaser, I. Millard, and A. Jaffri. RKBExplorer.com: A knowledge driven infrastructure for Linked Data providers. In *ESWC*, pages 797–801. Springer, 2008.
10. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pages 436–445, 1997.
11. O. Görlitz and S. Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *COLD*. CEUR (Vol 782), 2011.
12. S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C Recommendation, March 2013.
13. A. Hogan, J. Umbrich, A. Harth, R. Cyganiak, A. Polleres, and S. Decker. An empirical survey of Linked Data conformance. *JWS*, 14:14–44, 2012.
14. V. Janev and S. Vraneš. Applicability assessment of semantic web technologies. *Information Processing & Management*, 47(4):507–517, 2011.
15. G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *EDBT*, pages 29–47. Springer, 2004.
16. A. Langegger, W. Wöß, and M. Blöchl. A Semantic Web middleware for virtual data integration on the Web. In *ESWC*, pages 493–507. Springer, 2008.
17. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. DBpedia SPARQL benchmark – performance assessment with real queries on real data. In *ISWC*, pages 454–469. Springer, 2011.
18. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, January 2008.
19. B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *ESWC*, pages 524–538. Springer, 2008.
20. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on Linked Data. In *ISWC*, pages 601–616. Springer, 2011.
21. G. T. Williams. SPARQL 1.1 Service Description. W3C Recommendation, March 2013.