YASGUI: How do we Access Linked Data? [1]

Laurens Rietveld [a] and Rinke Hoekstra [a,b]
[a] *Department of Computer Science, VU University Amsterdam, The Netherlands*
E-mail: {laurens.rietveld,rinke.hoekstra}@vu.nl
[b] *Leibniz Center for Law, Faculty of Law, University of Amsterdam, The Netherlands*
E-mail: hoekstra@uva.nl

Abstract. The size and complexity of the Semantic Web makes it difficult to query. For this reason, accessing Linked Data requires a tool with a strong focus on usability. In this paper, we present YASGUI, a Web application for accessing the Semantic Web through SPARQL, that integrates live services and query management. We elaborate on the trade-offs that exist between these requirements, and discuss the restrictions inherent in application development for the Semantic Web. We identify typical SPARQL-specific tasks, and investigate how these relate to the actual usage SPARQL in general, and of YASGUI in particular.

Keywords: HCI, SPARQL, User Interface, Linked Data, Semantic Web, Tasks

1. Introduction

Developers who use traditional Web technologies are pampered with full-featured development tools such as in-browser debugging, integrated development environments, increasingly simple and lightweight services (RESTful APIs), and broad take up in industry. Semantic Web technologies have some catching up to do. The recent start of the W3C Linked Data Platform working group[1] is a good step in bringing triple-store querying closer to traditional RESTful APIs. However, the ingenuous developer wanting to have a first taste of Linked Data is scared away by austere clients for a rich but complex query language: SPARQL.

Indeed, several good RDF programming libraries exist, but uptake of these still relies on a good understanding of SPARQL and the underlying Semantic Web paradigm which can only be attained with simple, lightweight and user friendly clients for interacting with Linked Data. This observation holds for Semantic Web savvy developers as well: trying and testing SPARQL queries is often a cumbersome and painful experience. All who know the RDF namespace by heart raise their hands now! A related question that is hard to answer for many: "Where is that Linked Data?" Most will know the DBpedia endpoint URL, but can perhaps mention only a handful of other endpoints in total.

The first contribution of this paper is 'yet another SPARQL GUI' (YASGUI[2]), a SPARQL client that shows the added value of combining Web 2.0 and Semantic Web technologies [1,2] for providing a more gentle Linked Data interaction environment. We find that most existing SPARQL clients do not offer functionality that goes far beyond a simple HTML form (section 2). These implementations convey a rather narrow interpretation of what a SPARQL client interface should do: POST (or GET) a SPARQL query string to an endpoint URL. As a result, they currently offer only a selection of the features that we, as a community, could offer to both ourselves as well as new users of Semantic Web technology.

YASGUI is a web-based SPARQL client that can be used to query both remote and local endpoints. It integrates linked data services and web APIs to offer features such as autocompletion and endpoint lookup. It supports query retention – query texts persist across sessions – and query permalinks, as well as syntax checking and highlighting. YASGUI is easy to deploy

[1]This paper is an extended version of [21].
[1]See `http://www.w3.org/2012/ldp`

[2]See `http://yasgui.org`

locally, and it is robust. Because of its dependency on third party services, we have paid extra attention to graceful degradation when these services are inaccessible or produce unintelligible results.

The second contribution of this paper is an analysis of the types of things people use SPARQL for, and how this relates to SPARQL clients (and more specifically, YASGUI). The analysis consists of two parts. We combine several "taskonomies" [5,22,15,23] into a categorization of SPARQL-client specific tasks (Section 4). This analysis is then used to compare YASGUI to the state of the art, both through a questionnaire and by relating these tasks to typical patterns in SPARQL queries and server logs. We furthermore compare query characteristics between YASGUI and part of the USEWOD DBpedia query logs [3].

Structure of the paper

This paper is structured as follows. Section 2 provides an overview of the features present in the state of the art in SPARQL user interfaces. We then compare and explain the features and design considerations of YASGUI in section 3. In section 4, we present and discuss a taskonomy of tasks that are compatible with the potential interaction with SPARQL endpoints. Section 5 relates these tasks to the *reported* use of SPARQL by knowledgeable Semantic Web users, by means of questionnaire. We then continue to relate the tasks to the *actual* usage of SPARQL endpoints by analyzing and comparing the logs of YASGUI with that of DBPedia (section 6). We conclude in section 7.

2. State of the Art in SPARQL User Interfaces

The features of SPARQL clients can be categorized under three main headers, *syntactic* features (auto-completion, syntax highlighting and checking), *applicability* features (endpoint or platform dependent or independent) and *usability* (query retention, results rendering and download, quick evaluation). Table 1 lists twelve currently existing SPARQL clients – that range from very basic to elaborate – and depicts what features they implement. This section describes these features in more detail, and discusses whether and how the clients of Table 1 implement these features.

2.1. Syntactic Features

Most modern applications that feature textual input support some form of *auto-completion*. Examples are the Google website which shows an auto-completion list for your search query, or your browser which (based on forms you previously filled in) shows auto-complete lists for text inputs. One advantage of auto-completion is that it saves you from writing the complete text. Another advantages is the increase in transparency, as the auto-completion suggestions may contain information the user was not aware of. The latter is especially interesting for SPARQL, where users might not always know the exact prefix he/she would like to use, or where the user might not know all available properties in a triple-store. The only SPARQL interface that currently makes use of this functionality is the FLINT SPARQL Editor[3] (and indirectly, the SparQLed editor[8] based on the former), which uses auto-completion to suggest classes and properties.

Syntax highlighting is a common functionality for programming language editors. It allows users to distinguish between different properties, variables, strings, etc. The same advantage holds for query languages such as SPARQL, where you would like to distinguish between literals, URIs, query variables, function calls, etc. The only SPARQL editor that currently supports syntax highlighting is the FLINT SPARQL Editor, which uses the CodeMirror JavaScript library[4] to bring color to SPARQL queries.

Most Integrated Development Environments (IDEs) provide feedback when code contains syntax errors (i.e. *syntax checking*). Feedback is immediate, which means the user can spot syntax errors in the code without having to execute it. Again, such functionality is useful for SPARQL editing as well. Immediate feedback on a SPARQL syntax means the user can spot invalid queries without having to execute it on a SPARQL endpoint. The FLINT SPARQL editor supports syntax checking by means of a JavaScript SPARQL grammar and parser.

2.2. Applicability Features

There are only few clients who allows access to multiple endpoints. Most triple-stores provide a client interface, linking to that specific endpoint. They are *endpoint dependent*. Examples are 4Store [14], OpenLink Virtuoso [19], OpenRDF Sesame Workbench [7] and SPARQLer[5]. More generic clients are the Sesame2

[3] See `http://openuplabs.tso.co.uk/demos/sparqleditor`

[4] See `http://codemirror.net/`

[5] See `http://www.sparql.org/`

Table 1

SPARQL client feature matrix

Feature	4Store	OpenLink Virtuoso	SNORQL	SPARQLer	Sesame Workbench	Sesame2 Windows Client	Glint	Twinkle	SparqlGUI	SparQLed	Flint SPARQL Editor	YASGUI
Auto-completion	-	-	-	-	-	-	-	-	-	+ [a]	+ [a]	+ [b]
Syntax Highlighting	-	-	-	-	-	-	+	-	-	+	+	+
Syntax Checking	-	-	-	-	-	-	-	-	-	+	+	+
Multiple Endpoints	-	-	-	-	-	+	+	+	+ [c]	-	+/- [c]	+
Query retention	-	-	-	-	-	+	+	-	+	-	-	+
File upload	-	-	-	-	+	+/- [d]	-	+	+	-	-	- [e]
Platform independent	+	+	+	+	+	-	-	+	-	+	+	+
Results rendering	-	+/- [f]	+	+/- [f]	+	+/- [f]	+/- [f]	+/- [f]	+/- [f]	+	+	+
Results download	+	+	+	+	+	+	+	+	+	-	-	+

[a] Auto-completion of properties and classes available in the triple store

[b] Autocompletion of prefixes/namespaces. Autocompleting properties is implemented but not released yet. Autocompleting classes is a planned feature.

[c] Can deal with a limited number of endpoints, e.g. only CORS enabled ones.

[d] File upload requires a local triple store that implements the OpenRDF SAIL API, e.g. OpenRDF Sesame or OpenLink Virtuoso.

[e] File upload is a planned feature, using cloud triple-store services (e.g. dydra.com)

[f] The rendering does not use hyperlinks for URI resources.

Windows Client [7], Glint[6], Twinkle[7] and SparqlGUI[8]. Other applications fall somewhere in between. The FLINT SPARQL Editor only connects to endpoints which support cross-domain JavaScript (i.e. CORS enabled). This is a problem because not all endpoints are CORS enabled, such as FactForge, CKAN, Mondeca or data.gov. Other editors support only XML or JSON as query results, such as SNORQL[9] (part of D2RQ [4]), which only support query results in SPARQL-JSON format.

Platform (In)dependence increases the accessibility of a SPARQL client. The user can access the client on any operating system. Web interfaces are a good example: a site should work on any major browser (Internet Explorer/Firefox/Chrome), and at least one of these browsers is available for any type of common operating system. Examples are Virtuoso, 4Store and the Flint SPARQL Editor. Another example of multi-platform support is the use of a .jar file (e.g. Twinkle), as any major operating system supports Java and executing Java archives. Examples of single-platform applications are Sesame2 Windows Client and SparqlGUI: they require Windows.

[6]See `https://github.com/MikeJ1971/Glint`

[7]See `http://www.ldodds.com/projects/twinkle/`

[8]See `http://www.dotnetrdf.org/content.asp?pageID=SparqlGUI`

[9]See `https://github.com/kurtjx/SNORQL/`

2.3. Usability Features

Query retention allows for easy re-use of important or often used queries. This allows the user to close the application, and resume working on the query later. An example is the ‘ Query Book’ functionality of the Sesame Windows Client.

Quick evaluation or testing of a graph generated by the user should not require the hassle of installing a local triple-store Ideally, this functionality would be embedded in the SPARQL client application itself. Most applications requiring a local installation on the users computer support this feature, such as Twinkle. The Sesame Windows Client supports file uploads as well, though it requires a local triple-store which implements the OpenRDF SAIL API.

Query results (such as JSON or XML) for SELECT queries are often relatively difficult to read and interpret, especially for a novice. A *rendering* method which is easy to interpret and understand is a table. All applications except 4Store support the rendering of query results into a table. Because of the use of persistent URIs, we would expect navigable results for resources, e.g. in the form of drawing the URIs as hyperlinks. This feature is not supported by some applications, such as Virtuoso, Twinkle or SparqlGUI. SNORQL is an application with an elaborate way of visualizing the query results. Besides allowing the user to navigate to the page of the URI, the user can click on a link to browse the current endpoint for resources relevant to that URI.

Downloading the results as a file allows for better re-use of these results. A user might want to avoid running the same heavy query more than once, and use the results stored as a file instead. Additionally, the results of CONSTRUCT queries are often used in other applications or triple-stores Saving the user from needing to copy & paste query results clearly improves user experience as well. The only application that does not support the downloading of results, is the FLINT SPARQL editor.

Most of the clients described above are restricted to one simple task: accessing information behind a SPARQL endpoint. However, equally important to this task is assisting the user in doing so. This is something where all but one applications fail. Regrettably, the one interface with a user-friendly interface (FLINT SPARQL editor) falls short in the important feature of accessing all endpoints. We conclude that currently no single endpoint independent, accessible, user-friendly SPARQL client exists.

3. The YASGUI SPARQL Client

The overview of the preceding section shows us that current SPARQL clients fall short with respect to two important features: *accessibility* to Linked Data and *usability* of the client. The two most important goals of YASGUI (Figure 1) are:

1. support users in writing/executing SPARQL queries as much as possible, and
2. allow access to any SPARQL endpoint (either online or offline, regardless of response types such as XML or JSON).

Both goals bring inherent trade-offs for some features: tightly connecting a SPARQL interface with a single endpoint provides more freedom in usability features. For instance, calculate graph summaries offline for auto-completion purposes, such as SparQLed. However, such an interface would not be interoperable with other endpoints, thus conflicting with the first goal. In this section we discuss how we try to achieve both goals. We elaborate on the architecture, features, and design considerations of YASGUI, and compare them to the other clients.

3.1. Architecture

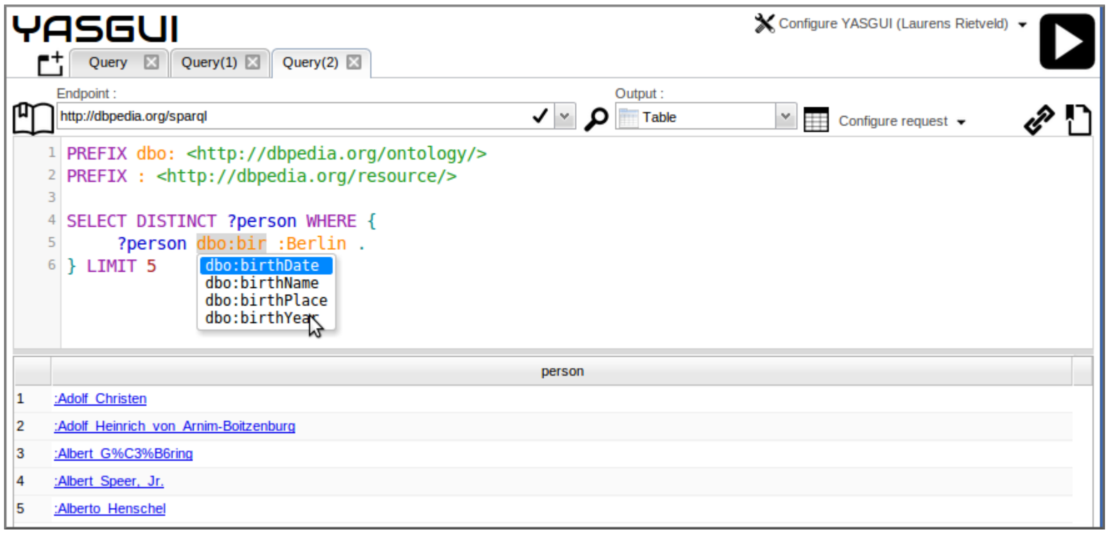
YASGUI was developed using the SmartGWT[10] and jQuery libraries.[11] It uses new HTML5 functionalities such as local storage and client-side generation of files. Some of the newest HTML5 functionalities are not supported by outdated browsers and Internet Explorer. This degradation is handled gracefully: access via an incompatible browser results in a notification to the user and disabled features (such as downloading of files, or client-side caching of large objects). The decision to use HTML5 is motivated by the increasing support of the standard by major browsers. The server-side part of YASGUI is responsible for part of the communication with external services and endpoints. Communication with SPARQL endpoints is done using the Jena library [13]. External services used by YASGUI are CKAN[12], Mondeca[13] and Prefix.cc[14] (see section 3.2), and bitly[15] (see section 3.4).

3.2. Syntactic Features

Two libraries provide support for syntax *highlighting* and *checking* in YASGUI: The CodeMirror JavaScript library, which is an extensive JavaScript library for highlighting code, and a JavaScript SPARQL grammar of the FLINT SPARQL Editor. Given this grammar, CodeMirror applies the highlighting to the SPARQL query. Additionally, CodeMirror provides a well documented API to parse and dissect the SPARQL query, useful for other YASGUI features such as prefix auto-completion. Both libraries are well documented, well maintained, extendable and easy to use. The existence of both libraries illustrate the availability of elaborate open source project, and the small amount of effort it takes to integrated them into an application.

Additionally, YASGUI uses Prefix.cc[16] to perform *auto-completion* of namespace prefixes: full names-

[10]See `http://www.smartclient.com/product/smartgwt.jsp`
[11]See `http://jquery.com/`
[12]See `http://semantic.ckan.net/sparql`
[13]See `http://labs.mondeca.com/endpoint/ends`
[14]See `http://prefix.cc/`
[15]See `http://bitly.com`
[16]See `http://prefix.cc/`



Endpoint : http://dbpedia.org/sparql

Output : Table

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX : <http://dbpedia.org/resource/>

SELECT DISTINCT ?person WHERE {
      ?person dbo:bir :Berlin .
} LIMIT 5
```

	person
1	:Adolf_Christen
2	:Adolf_Heinrich_von_Arnim-Boitzenburg
3	:Albert_G%C3%B6ring
4	:Albert_Speer,_Jr.
5	:Alberto_Henschel

Fig. 1. Screenshot of the YASGUI interface

pace URIs are completed as you type. We furthermore rely on the CKAN SPARQL endpoint[17] and Mondeca Endpoint Status endpoint[18] for endpoint URL auto-completion and search. The CKAN endpoint provides access to the CKAN datahub.io[19] catalogue of datasets, where the Mondeca endpoints filters these endpoints by accessibility. Users can either use a simple auto-completion combobox, or browse through / search in a list of endpoints in a table. This auto-completion matches the partially typed endpoint with the list of endpoints (and their descriptions).

Both endpoints have proven to be difficult to use and access for our purposes. The CKAN endpoint is rather unreliable in up-time, and the Mondeca endpoint often return syntactically invalid XML. In the implementation of YASGUI we try to handle both issues as gracefully as possible. The list of endpoints provided by CKAN or Mondeca is cached on the YASGUI server as well as by users browser. If YASGUI fails to retrieve the list in real time from either of the endpoints, we fall back to the cached results.

Another issue with CKAN (and to a lesser extent Mondeca) is the reward model for adding and maintaining the catalogue: there is little incentive for owners of a dataset to add it to CKAN, and even less incentive keep the information up to date (e.g. when the endpoint is down or moved). As a result, CKAN is cluttered with outdated information, and some endpoints are missing. This is partly compensated by Mondeca, which allows filtering by endpoints which are up, though incorrect or missing information still persists. The reward model employed by Prefix.cc is the opposite: the content is crowd-sourced (anybody can easily add prefix definitions), and voting is used to deal with conflicting prefix definitions. Users of prefix.cc have an incentive to keep the information up to date and as correct as possible. As a result, the information retrieved from prefix.cc is more reliable and usable than information from CKAN and Mondeca.

3.3. Applicability Features

As mentioned in section 2.2, client-side web applications such as the FLINT SPARQL Editor are *endpoint independent*, but only work for endpoints that enable Cross-Origin Resource Sharing (CORS).[20] To overcome this limitation, YASGUI includes a server-side proxy for accessing endpoints that do not support CORS. For endpoints which do support cross domain JavaScript, YASGUI executes the queries solely from the clients side via JavaScript. The only scenario where YASGUI fails to connect to an endpoint is where (1) a locally installed endpoint is unreachable from the web, (2) operating on a different port than YASGUI, and (3) CORS-disabled. Here, the YASGUI proxy is not able to access the client. Because of the CORS restriction, YASGUI is not able to access the endpoint via

[17]See `http://datahub.io`
[18]See `http://labs.mondeca.com/endpoint/ends`
[19]See `http://datahub.io`

[20]See `http://www.w3.org/TR/cors/`

JavaScript as well, as it is operating on a different port. We consider this issue to be minor: because the endpoint is installed locally, the user will have access to change its CORS settings, or even run the endpoint via a different port.

Other than dealing with the accessibility issues of CORS disabled sites, endpoint independent clients should support configurable requests. For instance, some endpoints may only support the XML results format, or allow the use of additional request parameters, such as the 'soft-limit' of 4Store. Such endpoints can only be used to their full potential if users are able to specify these additional arguments manually. Therefore, YASGUI supports the specification of an arbitrary number of request parameters for every endpoint.

Finally, YASGUI has to deal with the wide variety of possible *errors* returned by endpoints. The SPARQL protocol specifies what the endpoint request and response should look like, but leaves error handling unspecified: what HTTP error code should be sent by an endpoint, and how should error messages be communicated? As a result, triple stores come with various ways of conveying errors. Some endpoints return the error as part of an HTML page (with the regular 200 HTTP code), or as a SPARQL query result. Others only return an HTTP error code, where only some include a reason phrase together with the error code. The latter is a best practice for RESTful services. The absence of a standard, and the failure to adhere to best practices, makes a generic robust error handling solution messy and difficult to implement. Developing such a solution requires coding and testing by trial and error, and test queries on as many different endpoints as possible.

3.4. Usability Features

As Table 1 shows, most SPARQL clients support both *rendering* and *downloading* of query results to some extent. YASGUI does both as well. Users can render results either as a lightweight HTML table (for large number of results), an elaborate sortable/groupable table, or show the raw query results with syntax highlighting. Tables can be downloaded as CSV, where raw query results are available for download 'as is'. YASGUI supports the same functionality of SNORQL. Whenever the user clicks a resource from the query results, a new query opens and executes that shows information related to this resource.

YASGUI stores the application state, making this application state *persistent* between browser/user sessions: a returning user will see the screen as it was when she last closed the YASGUI browser page. Additionally, YASGUI supports bookmarking queries. This way users are able to re-use queries between user sessions, browsers, and computers.

Furthermore, YASGUI provides query *permalink* functionality. For a given query and endpoint combination, YASGUI creates a link that can optionally be shortened via the Bitly service.[21] Opening the link in a browser opens YASGUI with the specified query, endpoint and request arguments filled in. We believe this is a welcome feature for people working together with a need to share queries.

4. A SPARQL Taskonomy

As we have demonstrated in table 1, YASGUI has a richer feature set than the state-of-the-art in SPARQL client interfaces. And in fact, these features were added because we, the developers, think they are useful for our *own* work, and by transitivity to others working on the same types of projects in Semantic Web research. Secondly, we feel that a narrow interpretation of what a SPARQL client is supposed to support – i.e. "sending a query to a SPARQL endpoint" – does not do justice to the important role these clients play in Semantic Web development, use and uptake. The features of YASGUI touch upon many aspects associated with query writing, execution and management. We have not come across a comprehensive overview of the types of tasks SPARQL clients are used for in a broader Semantic Web context.

ISO/IEC 25010:2011 proposes a quality model for the evaluation of software[22], that "categorizes product quality properties into eight characteristics" (section 4.2): functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability and portability. The first of these, functional suitability, reflects what defines a system: to what degree does the system provide functions that meet certain stated or implied *needs*? In other words, with which *problems* in mind was the system developed, and to what extent does the system assist users in performing the tasks that solve these problems?

[21] See `http://bitly.com`

[22] See `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=35733`, the quality model in Section 4.2 is freely available online.

Like the Web [15], the range of application areas of Semantic Web technology has proven to be more versatile than expected when the technology was first standardized in the 1990s. An example is the growing importance of linked open government data. Instead of formulating strict functional requirements for YASGUI from the top down, we propose to work from the bottom-up, and analyze the use of YASGUI and other clients with respect to a list of typical tasks.

In this section we explore the types of tasks that pertain to the Semantic Web, and discuss how they may relate to the use of SPARQL. This results in a "SPARQL Taskonomy". The next section (section 5) shows the result of a survey amongst SPARQL users, and compares the position on the taskonomy of YASGUI with that of the other SPARQL clients that were discussed in section 3. What tasks is YASGUI deemed more suited for than the other clients?

4.1. Taskonomies

The distinction between task and domain knowledge, and the comparable attention for them that was prevalent in knowledge engineering of the eighties and early nineties (e.g. in KADS and CommonKADS, [5,22]) has largely disappeared into the background since the advent of ontologies. Inspired by this earlier research, van Harmelen et al. [23] studied the types of *applications* developed for the annual Semantic Web Challenge,[23] a challenge for the Semantic Web community to show the best of the Semantic Web. Similarly, Heath [15] developed a "taskonomy" (pun intended) for Semantic Web users based on a sizable body of research on tasks performed on the Web. Most importantly, he stresses the distinction between tasks and activities that represent *means* or *methods* from *ends* or *purposes*. Tasks that reflect *means* are too easily biased towards a specific technology or implementation. The work on Linked Data Patterns by Dodds and Davis [11], though useful in its own right, falls into the former category.

Breuker [5] categorizes eight *problem types* according to three major types: *synthesis*, *modification* and *analysis*. There is broad consensus about the distinction between the analysis and synthesis tasks. Synthesis involves the combining and creating entities, where modification refers to changing the behavior of the knowledge based system itself. In the later work of Schreiber et al. [22], this category is merged with the *synthesis* type. Analysis tasks take as input some data about a system, and produce some characterization of the system as output [5,22,10]. Here, the system is an "abstract term for the object to which the task is applied" [22]. Clancey [10] defines a system as a "a complex of interacting objects that have some process (I/O) behavior", e.g. a computer program, a university, an experimental procedure etc.

Breuker's major types are not meant to be disjoint: a task of a certain type may be composed of any number of subtasks that belong to other major categories. His problem types are to be used as an index to a collection of *problem solving methods* (PSM) [5,6,22]. Breuker [5] emphasizes the *problem* rather than the *task* for the same reason that Heath distinguishes *ends* from *means*: "indeed also in KADS tasks and PSM are closely related [..] but not keeping these apart may lead to a confusion of goals and means" [5, p. 60]. This is in line with the discussion by O'Hara and Shadbolt [18] of Chandrasekaran's generic task methodology [9].

These categories were identified in the context of expert systems design, at a time when the Web was still in its infancy: they are limited to the kinds of tasks one could expect an expert system to perform. In the following, we group the tasks identified in [15] and [23] into the *synthesis* and *analysis* categories, and a third "modern" dialectical category: *communication* (see Table 2).

4.1.1. Synthesis Tasks

Synthesis involves the combination of several entities into a new structure. The clearest example of synthesis are the *web-service composition* and *data integration* tasks of [23]. Service composition it is about combining a number of candidate services into a single composite service and involves aspects of both planning and design. Data integration constructs a "single, merged instance set, organised in a single, merged terminology" *Semantic enrichment* [23] involves the annotation of existing resources with concepts and terms from the Linked Data cloud.

The *asserting* task of [15] is more generic. It involves making "statements of fact or opinion available, with no discursive expectation". This can mean that new information is made explicit, that existing information is combined in a new way, or that prior statements are negated (retracted). What exactly constitutes the information in terms of Linked Data is intentionally left implicit.

[23]`http://challenge.semanticweb.org/`

Table 2
Tasks and activities from [5], [22], [15] and [23]

major type	Breuker [5]	Schreiber et al. [22]	Heath [15]	van Harmelen et al. [23]
synthesis	modeling design planning/reconstruction	modeling design planning assignment scheduling	asserting	web-service composition semantic enrichment data integration
modification	assignment (scheduling, configuration)			
analysis	prediction monitoring diagnosis assessment	prediction monitoring diagnosis assessment classification	locating exploring grazing monitoring evaluating	search browsing personalization & recommending web-service selection
communicating			sharing notifying discussing arranging transacting	

4.1.2. Analysis Tasks

Perhaps not surprisingly, the majority of analysis tasks listed by [15,23] are *search* related. We group the *search*, *personalization and recommending*, *browsing* and *web-service selection* tasks from [23] under this heading.

The *search* task is performed by applications that given a query and a dataset of instances, return a subset of these instances that match the query. [23] restrict the query to a concept description in some ontology O. It is a reformulation of the *classification* task of [22]: return all entities that meet certain criteria. *Web-service selection* involves a search for web services that meet certain criteria, on the basis of a partial description. *Browsing* is similar to these, but the queries used are more constrained, and "its output can be both a set of instances, or the immediate sub or super concepts of the input concept". *Personalization and recommending* takes a dataset plus a personal profile, and returns a smaller dataset based on this user profile.

Again, the search tasks of [15] are less loaded by technology. These are *locating*, *exploring*, *grazing*, *monitoring* and *evaluating*. *Locating* is looking for an object or chunk of information which is known/expected to exist, where *exploring* involves information gathering about something, or obtaining background knowledge of that thing. The *grazing* task is a more generic formulation of the *browsing* task of [23]. Heath defines grazing as moving speculatively between sources with no specific goal in mind.

The *evaluating* task is a form of *assessment* [5,22], it is about determining wether a particular piece of information is true. Finally, *monitoring* is a direct instantiation of the monitoring problem of [5,22]. It is the regular checking for changes to known resources, with the intention of detecting the occurrence and nature of changes.

4.1.3. Communication Tasks

This final category groups tasks where the main focus lies on communication. The following five tasks from [15] belong to this category: *sharing*, *notifying*, *discussing*, *arranging* and *transacting*. *Sharing* is making an object or chunk of information available to others, where *notifying* is informing others of an event in time or change of state. *Discussing* differs from sharing and notifying in that it is about the *exchange* of knowledge and opinions between persons or organizations. *Transacting* involves transferring money or credit between two parties. Finally, *Arranging* is the task of coordinating with others to ensure something will take place or will be possible at a certain time.

4.1.4. Discussion

The tasks of [23] and [15] are quite different. Where the former categorization sticks close to actual applications, and thus to technology, the latter is much more generic. The drawback of the application-based categories is that they are rather inflexible for accommodating new, unforeseen uses of the technology. Secondly, and perhaps as a consequence, the distinctions between the categories of [23] are sometimes too de-

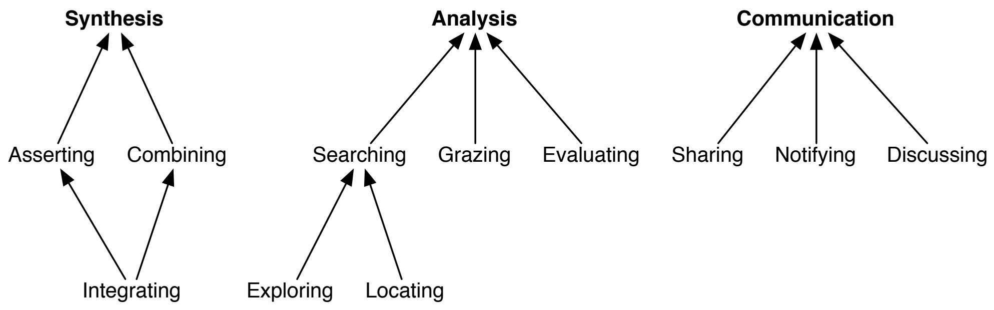


Fig. 2. A SPARQL Taskonomy

tailed and pragmatic. For instance, why distinguish a "web-service selection" task, and not a "hotel selection" task? The same may be said for the *personalization and recommendation* task.

Conversely, the downside of the generic categories in [15] is that they are sometimes a bit too generic, and it may prove to be hard to identify instantiations of these tasks in existing applications. For instance, the *asserting* task covers essentially any SPARQL update query, while the *data integration* task of van Harmelen et al. more closely captures the omnipresent *mashup*-style of combining information from multiple information sources.

Although we cannot dismiss them offhandedly, not all tasks we have listed can be linked to features of SPARQL clients. For instance, it is quite far fetched to imagine a client that actively supports the *transacting* and *arranging* communication tasks. On the other hand, the input/output of clients may well be used in *notification* and *discussing* tasks. Although it is possible to use a SPARQL client to *monitor* a triple store for changes, this is clearly functionality that is better suited for environments with less need for user intervention.

Taking these considerations into account, Figure 2 lists the final SPARQL taskonomy that we will be using to evaluate the usage of SPARQL clients in the following sections. Similar tasks, such as *exploring* and *locating* are grouped under a higher level task (*searching*). The distinction with *grazing* is that search implies an explicit formulation of a query by a user, whereas grazing is *speculative* browsing [15]. The *integrating* task is a combination of *asserting* and *combining* information.

5. Reported SPARQL Use

The SPARQL taskonomy presented in the previous section allows us to better understand how SPARQL endpoints are used, and how people use YASGUI. In this section, we explore the relation between the tasks, and the *reported* use of SPARQL by knowledgeable Semantic Web users. This information is gathered via an online questionnaire. Respondents were solicited via an email to various popular Semantic Web and Linked Data related mailing lists, as well as by means of a pop up screen in YASGUI. The following sub sections cover the design of our questionnaire, and the discussion of how our results relate to the usage of SPARQL endpoints and YASGUI.

5.1. Questionnaire setup

All questionnaire respondents receive questions on background information, the tasks they perform on endpoints, and the tools they use. Respondents who indicate that they use YASGUI receive additional questions on how they use YASGUI, and which features they consider important with respect to the taskonomy.

Table 3 shows the we questions presented to our respondents. First, respondents are asked *background* questions such as the country of residence, their occupation, and their field of study (only shown to students and academics). Additionally, we inquire as to their *familiarity* with several technologies and tools, such as SPARQL or ontologies. We complement this list with other non-Semantic Web concepts such as SQL and programming.

Next, we ask the respondent how frequently he or she uses tools for accessing SPARQL endpoints. The tools we present to the respondents is based on the list of tools in table 1. However, we grouped the list of endpoint interfaces directly shipped with endpoints, as respondents might have difficulty distinguishing between them. Additionally, we added two methods for accessing SPARQL endpoints which do not contain a graphical user interface: accessing SPARQL endpoints using *code* (e.g. java, python) or via *command-line* tools (e.g. curl).

The following question asks about the frequency by which the respondent performs certain tasks on SPARQL endpoints. The list we present is based on the previous section (more specifically, figure 2). Because some of these task are not necessarily exclusive (e.g. the act of asserting and combining might be performed simultaneously), we combine some of them. We present the following list to the user: Searching, Grazing, Evaluating, Integrating and Communicating. This should make the questionnaire as intuitive as possible, and avoid any confusion the respondent might have.

Further questions in the questionnaire are only shown when the respondent indicated that he or she uses YASGUI. We ask these respondents which tools and methods they used prior to using YASGUI. Whenever the previous answers of a respondent show that he or she uses a tool with the same *frequency* despite knowing YASGUI, we ask the respondents for their motivation for using that tool.

We furthermore ask the YASGUI-using-respondents to sort a list of YASGUI features by how well they support them in accomplishing a certain task. We ask this question only for the three tasks the respondent performs most frequently. The list of features we ask the respondent to re-arrange, is based on section 3:

- Query syntax highlighting
- Query syntax checking
- Generation of query permalinks
- Endpoint search and auto-completion
- Prefix auto-completion
- Query bookmarking
- SNORQL-type navigation
- Access to -all- endpoints
- Configurable requests
- Cross-platform operability

The final question was to re-arrange a set of potential *new* features by priority. This list of eight features is based on a backlog of feature requests from the Semantic Web community. This list of potential new features is:

- Visualizing DESCRIBE and CONSTRUCT results as a graph
- Visualizing results in charts (e.g. bar chart, pie chart, map)
- Class auto-completion
- Property auto-completion
- Query catalogue feature (add or tag queries, and search for queries)
- Offline functionality: opening YASGUI offline will still allow you to connect to a localhost endpoint
- Upload a small RDF file to query on
- Simple edit feature, where you can edit the query results, and execute the changes as insert query

5.2. *Questionnaire Results*

We received 39 responses to our questionnaire, out of which 12 respondents used YASGUI recently. This means that we do not have sufficient response to present statistically relevant conclusions. We can, however, present some of our results anecdotally.

Professional Background The majority of the respondents are academics with a background in Computer Science, and most respondents are familiar with the SPARQL protocol.

Tools By far the most common way of accessing SPARQL endpoints is via the client interface shipped with an endpoint. Other frequent tools (in decreasing popularity) are via code, command-line, YASGUI and SNORQL. All other tools are rarely used (i.e. by less than 10% of the respondents). Respondents indicate direct and quick access as the primary reason for using the standard web client of endpoints. The frequency of accessing SPARQL endpoints via code makes sense, as this is what the Semantic Web is particularly suited for: allowing machine agents to communicate with SPARQL endpoints. Programming such an agent requires accessing these endpoints using code. The use of command line relates to the previous, as questionnaire respondents indicate they use command line for programming purposes as well, such as post processing the SPARQL results in bash, validating SPARQL endpoint server responses, and general scripting.

Task Frequency Figure 3 shows the statistics for how often our respondents perform each of the tasks, broken down against the type of user. Figure 3a shows the tasks frequency distribution of all questionnaire respondents, where figure 3d only shows results for respondents who have indicated that they use YASGUI. Figure 3c shows how often these respondents actually *use* YASGUI for performing the tasks. The final figure (figure 3b) shows the task frequency for respondents who are unfamiliar with YASGUI.

As figure 3a shows, respondents perform the searching task most frequently, followed by grazing and inte-

Table 3

Questionnaire setup

#	Topic of question	Answers	Note
	General questions asked to all respondents		
1	Country of residence	List of 272 countries	
2	Occupation/Profession	Academic, Student, Industry, other (namely)	
3	Field of study (multiple responses possible)	Natural Science, Social Science, Computer Science, Humanities, other (namely)	Only shown to students / academics
4	Familiarity with SPARQL, ontologies, programming, SQL and Linked Data	Scale from 1 (unfamiliar) to 5 (familiar)	
5	Frequency of usage of tools/methods (see section 2) for accessing SPARQL endpoints (in the last 3 months)	5-scale frequency answers [a]	
6	Frequency of performing tasks (see Fig 2)	5-scale frequency answers [a]	
	Questions shown to respondents who use YASGUI		
7	Frequency of usage of tools/methods *prior* to using YASGUI	5-scale frequency answers [a]	
8	Frequency of performing tasks using YASGUI (see Fig 2	5-scale frequency answers [a]	Only for tasks the respondent performs monthly
9	Motivation for using a tool	open question	Asked for every tool the respondent uses alongside YASGUI
10	Priority of features related to a given task	List of YASGUI features which the respondents needs to drag/drop and rearrange	Asked separately for the three most frequently performed tasks by the respondent
11	Importance of new features	List of new features which the respondent needs to drag/drop and rearrange	

[a] (almost) never, 1-2 times a month, 3-5 times a month, 6-15 times a month, 16+ times a month

grating. The frequency of searching and grazing shows that SPARQL endpoints are primarily used for information retrieval related purposes. Both these tasks are performed most often with YASGUI as well (figure 3c). Sharing and evaluating are performed least frequent.

The frequency distribution between the respondents unfamiliar with YASGUI (figure 3b) and the respondents familiar with YASGUI (figure 3c) is interesting: integrating, searching, grazing and communicating are performed more frequent by the latter. In other words, users familiar with YASGUI have a different usage behavior of SPARQL endpoints than users unfamiliar with YASGUI. However, do the former use other tools in performing their task, or do they use YASGUI? The difference between figure 3c and figure 3d answers this question: the tasks are performed less frequent *using* YASGUI, indicating that respondents use other tools instead.

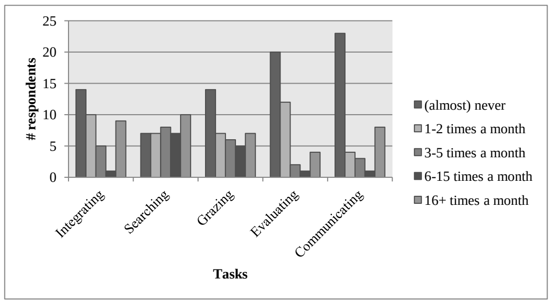
Features Unfortunately, the data that resulted from the question where we asked YASGUI users to order features by usefulness, per task, is too sparse to draw any definitive conclusions. However, if we look only at the features themselves, and aggregate the data across the tasks, we do see some interesting results.

Access to *all* endpoints is considered the most important feature. Prefix auto-completion, SPARQL syntax checking, SPARQL syntax highlighting and SNORQL-type navigation follow in priority. Respondents consider the features of query bookmarking, query permalinks, cross-platform operability, endpoint search and configurable request to be less important.

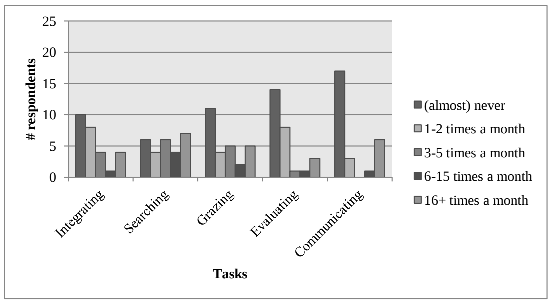
Potential Features Users of YASGUI are most interested in property and class *autocompletion*, and the ability for YASGUI to work in *offline* mode. Especially the latter is interesting, as the request for this feature indicates YASGUI is regularly used to access endpoints that are installed locally. Using a locally installed endpoint indicates that such a user is managing his or her *own* triple-store.

6. Actual SPARQL Use

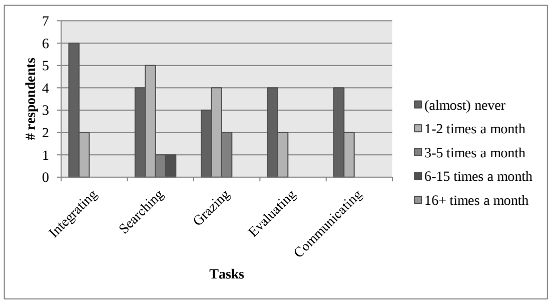
The previous section discussed the tasks users reportedly execute on SPARQL endpoints, and how this relates to YASGUI and other SPARQL clients. This



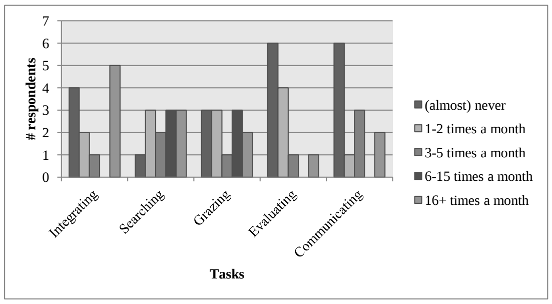
(a) All questionnaire respondents



(b) Respondents unfamiliar with YASGUI



(c) Tasks performed *using* YASGUI



(d) Respondents familiar with YASGUI

Fig. 3. **Task frequencies**

does not provide a complete overview though: we only had a limited number of respondents familiar with YASGUI, and using the questionnaire as an instrument does not provide a detailed view on the usage of YASGUI (e.g. what button is pressed how many times). For these reasons, this section explores whether we can relate our YASGUI usage logs to the tasks from section 4, and if so, about the effect of YASGUI on the behavior of users, compared to other clients.

We first introduce the *data sources* we analyzed to determine SPARQL usage and task execution (section 6.1), followed by a description of the *analysis method* in (section 6.2). Section 6.3 presents the patterns we observe as a result of the analysis, followed by some general observations in section 6.4.

6.1. Data Sources

We use *Google Analytics*[24] to log the actions of YASGUI users. These actions include the *queries* a user executes, the *endpoint* they use, and more general information such as (an estimate of) the users *location*, the *time*, etc. However, we do not track the actions of all users: every user is presented with an opt-out form in which users may choose to disable Google Analytics logging completely, or to disable Google Analytics for logging of endpoints and queries only. As a result, the YASGUI statistics described in this section are incomplete: 65% of the users specified everything could be logged, 5% disabled logging of endpoints and queries only, where the remainder disabled logging altogether. These logs show that YASGUI received 1124 unique visitors over the past 6 months, and 7707 queries were executed against 294 different endpoints.

[24]See `http://www.google.com/analytics/`

Secondly, we can use the *Bitly* service as a source for user logs as well. YASGUI uses Bitly to convert long query permalinks to short-links. This feature is optional: users may use the long URL as is, or they can convert it to a short link. This means that not all use of the query permalink functionality is logged, but only the query permalink functionality in combination with Bitly.

Google Analytics and Bitly provide us with the data to analyze the usage of YASGUI and link these to our user tasks. However, can we relate this to the usage of other clients? Does YASGUI invite users to write different or more complex queries? To achieve this, we compare YASGUI queries with those listed in the DBpedia 3.8 server logs published via the USEWOD workshop [3]. These DBpedia logs are anonymized, and contain all the HTTP (SPARQL) requests send to the DBpedia endpoint. Unfortunately, the YASGUI logs only maintain explicit links between endpoint and query since May 2013. Therefore, we were only able to retrieve YASGUI DBPedia queries from this point onwards.

6.2. Analysis Method

First, we extract relatively simple information such as the use of SPARQL keywords (e.g. DESCRIBE, ASK, SELECT, etc.), and relate these to the taskonomy from section 4:

- *Communicating* is associated with the use of the URL shortening service.
- *Evaluating* can be done via ASK and DESCRIBE queries, but also by means of very simple SELECT style queries.
- *Integrating* data is most typically performed through CONSTRUCT and INSERT queries.
- *Searching* is most strongly associated with SELECT queries, though DESCRIBE queries can be used as well.
- *Grazing* is often done through some form of (HTML) rendering of query results by services such as SNORQL and Pubby. This means that both SELECT and DESCRIBE queries may be used. We have identified a separate category for these SNORQL-type SELECT queries (see below).

See 4 for a quick overview of the relation between tasks and simple query types.

A deeper analysis of the structure of the queries allows for more fine-grained links between queries and tasks, though this fine-grained level may make it difficult to detect all queries matching a certain task. Note that we want to both analyze YASGUI usage *in general* and *in comparison* to other clients. The latter is done by comparing queries against DBpedia via YASGUI, to the general DBpedia server logs. DBpedia query logs have been subject to analysis before [12,20], and in this paper, we analyze all logged queries in a similar fashion.

To analyse the difference in *complexity* between the query sets, we use the same method as [12]:

Triple pattern structure The number of triple patterns used in queries, as well as the structure of these triple patterns. Each element in a triple can be a variable (V), or a constant (C). Using this abstraction, we can classify `[] rdf:type ?object` as `V C V`.

Joins When two triple patterns have one variable in common, the query engine would need to join both. We use the method described in [12] to calculate the number of joins per query, and the type of join. For each triple pattern containing a subject (S), predicate (P) and object (O), there are 6 possible join types: SS, PP, OO, SP, SO, PO. And example of two triple patterns where an SO join takes place is: `:NY :hostsConference ?conference` and `?conference rdfs:label ?name`.

The idea is that more complex queries will show more triple patterns and more joins.

The SNORQL-type queries we alluded to above are identified by matching queries to a *pattern* that is indicative of grazing behavior. For instance, in YASGUI, these queries are executed when clicking on a resource in your query results. Because the results of the SNORQL query are browsable as well, this behavior typically emerges when users speculatively navigate between resources. The SNORQL pattern contains one $UNION$, three projection variables, and two triple patterns (where the first triple contains a projection variable in the predicate and object position, and the second triple contains a projection variable in the subject and predicate position). An example of such a SNORQL-type query is shown below:

6.3. Results

6.3.1. Permalinks

The Bitly logs (taken from late March until early August) show the URL shorten service is used for 50

Table 4
Query properties

	YASGUI	**YASGUI (DBpedia)**	**DBpedia logs**	**task**
# syntactically valid queries	6507	372	1.466.287	
# time URL shortener is used	50	20	N/A	Communicating
SELECT queries	92.08%	98.66%	93.62%	Searching / Grazing / Evaluating
DESCRIBE queries	0.81%	0.81%	0.00%	Searching / Grazing / Evaluating
ASK queries	0.15%	0.00%	6.26%	Evaluating
CONSTRUCT queries	6.96%	0.54%	0.12%	Integrating
INSERT queries	0.00%	0.00%	0.00%	Integrating
SNORQL-type queries	6.63%	8.33%	0.01%	Grazing

SPARQL Query 1 Example SNORQL query

```
SELECT ?prop ?hasValue ?isValueOf
WHERE {
  { <uri> ?prop ?hasValue }
  UNION
  { ?isValueOf ?prop <uri> }
}
```

queries, which are clicked on (by others) a total 62 times. In other words, this feature is seldom used. Using the Bitly service is the most straight-forward way in YASGUI to share a query (compared to for instance copy/pasting a query and endpoint URL in an email). This indicates that the task communicating is uncommon. This corresponds with our questionnaire, where $2/3$ of the respondents indicated that they never perform the task of communicating.

6.3.2. *Queries*

Our queries logs show a total of 7707 queries. However, after filtering invalid queries using the Jena[25] query parser, this number drops to 6507 queries. When we remove duplicate queries from this query set, 3447 queries remain. Table 4 shows the YASGUI usage statistics when we relate query types to the SPARQL taskonomy.

SELECT The majority of queries executed via YASGUI are SELECT queries. However, linking these queries to a specific task is far from trivial. SELECT queries are suited for typical information retrieval tasks such as searching and grazing. Even evaluation tasks (for which the ASK keyword would be a sensible solution) may use the SELECT keyword instead. That being said, our questionnaire shows searching as the most often used task, which does correspond with the prevalence of the SELECT keyword usage.

ASK and DESCRIBE ASK queries amount to 0.15% of all YASGUI queries. As mentioned before, this keyword is a sensible one to use for the task of evaluating. However, the low use of this keyword does not correspond to the frequency our questionnaire respondents perform this task (half of them indicate performing this task at least once a month). We believe this shows that respondent prefer the more common SELECT keyword instead, as this suits the purpose of evaluating as well. Instead of returning a boolean value when using an ASK query, the user may evaluate the query results from the SELECT query as-is. Because users are probably more used to executing SELECT queries, only a few of them will opt for an ASK query. In other words, accurately mapping the Evaluation task to a specific set of SPARQL queries is not possible.]

The DESCRIBE query type meets the same fate, with a score 0% in the YASGUI logs: users apparently prefer to write SELECT queries instead.

CONSTRUCT CONSTRUCT queries are relatively popular as well with 6.96%. Together with INSERT queries (0.00%), these keywords correspond to the integrating task. This task is relatively common among questionnaire respondents as well: $2/3$ of them perform this task monthly. However, several questionnaire respondents indicated that they do not directly use SPARQL for these tasks, but rather prefer tools such as Protege [17] instead. The relatively high number of CONSTRUCT queries, as well as the large number of respondents performing this task, shows that SPARQL interfaces should not only focus on the retrieval of information. They should support the user in integrating information as well.

[25]See http://jena.apache.org/

SNORQL Table 4 shows the number of SNORQL-type queries as well. Our results[26] show these queries are relatively popular: 6.63% of the queries executed on YASGUI have the structure of a SNORQL query (indicating they perform the grazing task), which we consider a relatively large number.

6.3.3. Query Comparison

Here, we try analyze and compare the usage of YASGUI to the usage of other clients. We do so by comparing the properties of DBpedia queries executed via YASGUI, with the query properties from the DBpedia server logs. The YASGUI DBpedia query set contains 372 queries, where the DBpedia queries from the server logs (all USEWOD server logs from DBpedia 3.8) contains 1.4 million queries. Although the number of YASGUI DBpedia queries is only a fraction of the number of queries from the DBpedia server, we believe this number provides enough queries to analyze the differences between both query sets.

Query Types Table 4 shows a number of interesting differences between both query sets. In general, most query types (such as the number of SELECTs) correspond roughly. However, the number of ASK queries greatly differs between both: 0% in the YASGUI DBpedia logs, and 6.26% in the DBpedia server logs.

Additionally, the number of SNORQL-type queries greatly differs, as this number contributes to 8% of the YASGUI DBpedia queries, and 0.01% of the DBpedia log queries. In the previous section we made a connection between this type of queries and the grazing task. For DBpedia however, counting the number of SNORQL-type queries only retrieves a subset of the people performing this task. This is because several services, such as the DBpedia faceted browser[27] use DESCRIBE queries instead.

Nevertheless, even when enumerating the SNORQL-type queries (0.01%) and DESCRIBE queries (0.00%), the number of queries related to Grazing remains marginal compared all queries executed on DBpedia. This shows a striking difference between how often the grazing task seems to be performed on the DBpedia endpoint, and with respect to the times this task is performed using YASGUI. A possible reason for this difference is a low awareness of tools such as SNORQL and the faceted browser. Where YASGUI has the SNORQL feature integrated in the application, users accessing the DBpedia endpoint would need to use a search engine or read the DBpedia wiki to find an application supporting the Grazing task on DBpedia. In other words: combining such features in one single application increases the use of these features.

Query Complexity Table 5 shows striking *structural* differences as well. The YASGUI DBpedia logs show a larger number of queries with at least 1 join than the DBpedia server logs. Both the frequency of the triple pattern types differ greatly between the query sets, as well as the number of queries with at least 1 join. In other words, the structure of the queries from the DBpedia server logs are quite different from the structure of the DBpedia queries executed via YASGUI.

However, using this information to compare the usage of YASGUI with the usage of other clients is virtually impossible. The very probable reason for these differences is that DBpedia contains queries executed by *humans* as well as *machine* agents. And, one cannot differentiate between user and agent queries. Because both humans and machine agents use the HTTP protocol, it is virtually impossible to accurately distinguishing both at the HTTP request level. [16] recognizes this same problem as well.

That we cannot distinguish with certainty is exactly why the YASGUI query logs are interesting. Especially in cases where the research problem is targeted towards human users (as we do), the YASGUI logs provide a much more accurate reflection of endpoint usage by human users. A first step in detecting differences in the queries between machines and humans, is knowing what type of queries humans execute. YASGUI offers this first step. To our knowledge, YASGUI is currently the only service containing a diverse (query-wise, as well as endpoint-wise) set of SPARQL queries solely made by humans.

6.4. General Observations

The results described above related the usage of YASGUI and DBpedia to our tasks from section 4. However, unrelated from these tasks, are two other interesting observations we want to elaborate.

First, we analyzed the endpoint usage of YASGUI users. Where the previous section showed the actions users perform on YASGUI, the endpoint usage would show what endpoints users perform these on. To filter typographic errors, we reduce the list of 294 endpoints to a list of endpoints which only contain those

[26]The SNORQL feature was added in the end of May. Therefore, the query set to calculate the number of SNORQL-type queries on has a smaller sample size (from May onwards)

[27]See `http://dbpedia.org/fct/`

Table 5

Query complexity

	YASGUI	YASGUI (DBpedia)	DBpedia logs
Queries with 1 or more joins	47.20%	7.80%	12.32%
Queries with at least 1 V C C triple pattern	49.29%	20.97%	30.89%
Queries with at least 1 C V V triple pattern	8.71%	14.62%	22.8%
Queries with at least 1 V V C triple pattern	52.10%	13.77%	16.69%
Queries with at least 1 C C V triple pattern	10.97%	1.48%	9.95%
Queries with at least 1 V V V triple pattern	8.58%	15.04%	9.12%
Queries with at least 1 C V V triple pattern	9.31%	14.62%	0.71%
Queries with at least 1 V C V triple pattern	40.96	11.86%	0.71%
Queries with at least 1 C V C triple pattern	0.14%	0.00%	3.19%
Queries with at least 1 C C C triple pattern	2.03%	0.27%	0.04%

Table 6

YASGUI endpoint usage

	Relative to # endpoints	Weighted by #queries executed on endpoint
CKAN endpoints	18%	56%
Inaccessible endpoints	58%	32%

where more than 1 query was executed on. This results in a list of 185 endpoints. For each of the endpoints in this filtered list, we check whether this endpoint is in the CKAN dataset catalogue, and whether this endpoint is accessible. The results are shown in table 6. Results show 18% of these endpoints occur in the CKAN dataset catalogue, where 58% of these endpoints are inaccessible from the internet. When we weight these percentages using the number of queries executed on these endpoints, we observe 56% of the endpoints occurs in the CKAN dataset catalogue, and 32% of the endpoints are inaccessible from the internet. The difference between the weighted and unweighted percentages is mostly due to the large number (43%) of queries executed on DBpedia (which occurs in CKAN, and is obviously reachable).

These results show that a relatively large number of endpoints accessed via YASGUI are not listed in the CKAN dataset, and are not reachable from the internet. About $1/3$ of all executed queries are targeted to localhost or intranet SPARQL endpoints. We believe these results indicate that a large part of our queries are executed against non-public endpoints and endpoints of which the data is still under construction (e.g. installed locally). This corresponds to the questionnaire results, where the respondents were particularly interested in offline functionality for YASGUI (something only suitable to situations where an endpoint is installed locally).

Our second other observation involves the differences in query complexity between the queries executed via YASGUI, and the DBpedia queries. Table 5 shows almost half of all queries executed via YASGUI contain 1 or more joins. The DBpedia queries however (taken from both the YASGUI DBpedia queries, as well as the server logs) show at most 12.32% of the queries contain 1 or more join. The different triple pattern structures show similar large differences. (e.g. the VVC triple pattern: 52.10% vs. max 16.69%, respectively). These difference might be caused by the type of users accessing DBpedia, or the structure of the DBpedia graph. These numbers ask for a more thorough analysis in future work.

7. Conclusion and Future Work

The size and complexity of the Semantic Web make it difficult to query, and requires tools with a strong focus on usability. In this paper we presented the state of the art in SPARQL user interfaces, and showed most of these are rather austere clients with little focus on usability, and feature completeness. Most striking is that their functionality is largely complementary: we have the SNORQL client for associative browsing, the

FLINT SPARQL editor for highlighted queries, and other tools whose major selling point is access to *any* SPARQL endpoint. This large collection of tools, each with their own specific 'area of expertise', makes it hard for users to find and use the right tool for their task. Increasing user accessibility to the Semantic Web would require one single tool combining such features as possible.

This is why we introduced the YASGUI SPARQL client, a SPARQL client that shows the added value of combining Web 2.0 and Semantic Web technologies for providing a more gentle Linked Data interaction environment. It is a web-based SPARQL client that can be used to query both remote and local endpoints. It integrates linked data services and web APIs to offer features such as autocompletion and endpoint lookup. It supports query retention – query texts persist across sessions – and query permalinks, as well as syntax checking and highlighting.

In section 4, we stated that evaluating a Semantic Web tool such as YASGUI, starts with a *functionality analysis*: does the tool adequately support the tasks for which it was designed? The question we asked ourselves is to what extent do we *know* what tasks SPARQL clients are used for? Limiting the functionality analysis to just the interaction with a SPARQL endpoint over the HTTP protocol is a too broad criterion. Clearly, all clients discussed in section 2 meet this requirement. We furthermore argued, that this holds even for the Semantic Web itself. The quest for the Semantic Web killer app is still ongoing, and one can wonder whether such a thing can even be found.[28] Rather, as happened to Web technology in general [15], new uses of the technology will emerge and change our views. The shift of emphasis from description logics and OWL to Linked Data over the past years fits this picture.

With these considerations in mind, we propose a 'taskonomy' of types of tasks that we feel can be associated with the use of SPARQL endpoints (see figure 2). We used the taskonomy to come to grips both with the *reported* usage of tools for accessing SPARQL endpoints, and the *actual* usage. For the former, we conducted a questionnaire-based survey (section 5), for the latter we analyzed query logs of both YASGUI and the USEWOD logs for DBPedia.

[28]See `http://databanker.com/2013/03/05/the-myth-of-the-killer-app/` and `http://beust.com/weblog2/archives/000474.html` for insightful discussions on whether or not a killer app exists.

Our analysis shows that users of SPARQL *do* recognize the tasks in the taxonomy as relevant to their work with SPARQL. Classical information retrieval tasks such as *searching* and *evaluating* are by far the most often executed tasks on the Semantic Web, and should deserve attention in relation to the usability features of SPARQL clients. The questionnaire shows that Semantic Web users perform *integrating* tasks frequently as well, but our respondents do not often use SPARQL clients for this task. Interestingly, our first results show that users of YASGUI perform the integrating task more often than users of other clients, but they do not use YASGUI for it. The logs show a similar picture, only few integrating-related DESCRIBE queries are executed via YASGUI. Clearly, a feature for feeding the results of e.g. CONSTRUCT queries back to the same or a different triple store could be quite beneficial to the YASGUI user community.

When take a closer look at the *complexity* of queries formulated via YASGUI, compared to that of DBPedia, we cannot draw any real conclusions. Even though we can e.g. see in Table 5 that the CVV and VCV triple patterns are overrepresented in YASGUI, the hypothesis that a richer user interface for SPARQL queries would support more complex querying cannot be tested. This is for two reasons. Firstly, we have too little knowledge of what the instantiations of tasks look like at the level of actual SPARQL queries. And secondly, the DBPedia logs intermix queries posed by people directly with queries that are fired by Semantic Web applications. This is like comparing arts and crafts with industrial products. The area of Human-Computer Interaction and Semantic Web needs more specific and detailed query logs. Despite millions of (semi-)public queries available as server logs, we have no way of knowing *how* and *for what purpose* people create queries. We feel that the YASGUI logs are a promising first start.

Acknowledgements

This work was supported by the Dutch national program COMMIT/.

References

[1] Anupriya Ankolekar, Markus Krötzsch, Thanh Tran, and Denny Vrandečić. The two cultures: Mashing up Web 2.0 and the Semantic Web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(1):70–75, February 2008.

[2] Robert Battle and Edward Benson. Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST). *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(1):61–69, February 2008.

[3] Bettina Berendt, Laura Hollink, Markus Luczak-Rösch, Knud Möller, and David Vallet. Proceedings of USEWOD2013 - 3rd international workshop on usage analysis and the web of data. In *10th ESWC - Semantics and Big Data, Montpellier, France*, 2013.

[4] Christian Bizer and Andy Seaborne. D2rq - treating non-rdf databases as virtual rdf graphs. *World Wide Web Internet And Web Information Systems*, page 26, 2004.

[5] Joost Breuker. A suite of problem types. In *CommonKADS library for expertise modelling: reusable problem solving components*, volume 21 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 1994.

[6] Joost Breuker. Indexing problem solving methods for reuse. In *Knowledge Acquisition, Modeling and Management*, pages 315–322. Springer, 1999.

[7] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information, 2001.

[8] Stephane Campinas, Thomas E Perry, Diego Ceccarelli, Renaud Delbru, and Giovanni Tummarello. Introducing rdf graph summary with application to assisted sparql formulation. In *Database and Expert Systems Applications (DEXA), 2012 23rd International Workshop on*, pages 261–266. IEEE, 2012.

[9] Balakrishnan Chandrasekaran. Design problem solving: A task analysis. *AI Magazine*, 11(4), 1993.

[10] William J. Clancey. Heuristic classification. *Artificial Intelligence*, 27(3):289 – 350, 1985.

[11] Leigh Dodds. Twinkle: A sparql query tool.

[12] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. In *1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) at the 20th International World Wide Web Conference (WWW 2011), Hydebarabad, India*, 2011.

[13] Michael Grobe. Rdf, jena, sparql and the 'semantic web'. In *Proceedings of the 37th annual ACM SIGUCCS fall conference*, SIGUCCS '09, pages 131–138, New York, NY, USA, 2009. ACM.

[14] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The design and implementation of a clustered RDF store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, pages 94–109, 2009.

[15] Tom Heath. A taskonomy for the semantic web. *Semantic Web*, 1(1):75–81, 2010.

[16] Knud Möller, Michael Hausenblas, Richard Cyganiak, and Siegfried Handschuh. Learning from linked open data usage: Patterns & metrics. 2010.

[17] Natalya F Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W Fergerson, and Mark A Musen. Creating semantic web contents with protege-2000. *Intelligent Systems, IEEE*, 16(2):60–71, 2001.

[18] Kieron O'Hara and Nigel Shadbolt. Locating generic tasks. *Knowledge Acquisition*, 5(4):449 – 481, 1993.

[19] Openlink Virtuoso. Universal server platform for the real-time enterprise, 2009.

[20] Francois Picalausa and Stijn Vansummeren. What are real sparql queries like? In *Proceedings of the International Workshop on Semantic Web Information Management*, page 7. ACM, 2011.

[21] Laurens Rietveld and Rinke Hoekstra. Yasgui: Not just another sparql gui. In *Proceedings of the Workshop on Services and Applications over Linked APIs and Data (SALAD2013)*, 2013.

[22] Guus Schreiber. *Knowledge engineering and management: the CommonKADS methodology*. the MIT Press, 2000.

[23] Frank Van Harmelen, Annette Ten Teije, and Holger Wache. Knowledge engineering rediscovered: towards reasoning patterns for the semantic web. In *Foundations for the Web of Information and Services*, pages 57–75. Springer, 2011.