A Human-friendly Query Generation Frontend for a Scientific Events Knowledge Graph

Said Fathalla\textsuperscript{1,3}, Christoph Lange\textsuperscript{1,2}, and Sören Auer\textsuperscript{4,5}

\textsuperscript{1} Smart Data Analytics (SDA), University of Bonn, Germany
\{fathalla,lange\}@cs.uni-bonn.de
\textsuperscript{2} Fraunhofer IAIS, Germany
\textsuperscript{3} Faculty of Science, Alexandria University, Egypt
\textsuperscript{4} Computer Science, Leibniz University of Hannover, Germany
\textsuperscript{5} TIB Leibniz Information Center for Science and Technology, Hannover, Germany
soeren.auer@tib.eu

Abstract. Recently, semantic data have become more distributed. Available datasets increasingly serve non-technical as well as technical audiences. This is also the case with our EVENTSKG dataset, a comprehensive knowledge graph about scientific events, which serves the entire scientific and library community. A common way to query such data is via SPARQL queries. Non-technical users, however, have difficulties with writing SPARQL queries, because it is a time-consuming and error-prone task, and it requires some expert knowledge. This opens the way to natural language interfaces to tackle this problem by making semantic data more accessible to a wider audience, i.e., not restricted to experts. In this work, we present SPARQL-AG, a front-end that automatically generates and executes SPARQL queries for querying EVENTSKG. SPARQL-AG helps potential semantic data consumers, including non-experts and experts, by generating SPARQL queries, ranging from simple to complex ones, using an interactive web interface. The eminent feature of SPARQL-AG is that users neither need to know the schema of the knowledge graph being queried nor to learn the SPARQL syntax, as SPARQL-AG offers them a familiar and intuitive interface for query generation and execution. It maintains separate clients to query three public SPARQL endpoints when asking for particular entities. The service is publicly available online and has been extensively tested.

Keywords: Scientific events · SPARQL endpoint · Query builder · User Interaction · EVENTSKG dataset

1 Introduction

Nowadays, large amounts of semantic data have become widely available on the Web. This plethora of semantic data and the wide range of domains this data belongs to make it difficult to query this data. In addition, querying such data is a ponderous process, not only because of the syntax barrier, but mainly because of data heterogeneity and diversity. Semantic data is queried by means of the
widely-adopted W3C-standardized SPARQL query language [6]. SPARQL queries are executed against SPARQL endpoints, i.e., standardized query interfaces for semantic data stores. The advantages of SPARQL come from its expressivity and scalability, however, people spend a large part of their time to learn how to write a SPARQL query to fulfill their needs and, in many cases, they fail. In this article, we present SPARQL-AG, a semantic web frontend that assists users in generating SPARQL queries for querying the EVENTSKG knowledge graph [7], a comprehensive knowledge graph for scientific events in computer science. The rationale to develop SPARQL-AG is to help potential semantic data consumers, including both SPARQL experts and non-experts, by automatically generating SPARQL queries, ranging from simple to complex ones, using an interactive web interface. It helps SPARQL experts by reducing the time required to write queries by modifying the generated query (modify-before-execution), i.e., removing the need to write the query from scratch. The generated query is displayed in a readable way to make it easier to understand when a modification is needed before execution. The ultimate goal behind this work is to widen the access to semantic data available on the Web by making it easier to generate and execute SPARQL queries without prior knowledge of neither the schema of the data being queried nor the SPARQL syntax.

The architecture of SPARQL-AG is composed of five components: user interface, components selection, query composer, SPARQL clients manager, and query executor. This architecture integrates aspects of four research paradigms: query building (QB), semantic search (SS), human-computer interaction (HCI), and SPARQL query federation. Most of the SPARQL 1.1 specification [6] is covered, such as optional graph patterns, filters, aggregations, restricting aggregations, ordering, and limiting the number of results. SPARQL-AG maintains three SPARQL clients to query three public SPARQL endpoints (DBpedia SPARQL endpoint⁶, the Scientific Events Ontology (SEO) SPARQL endpoint⁷, and the EVENTSKG SPARQL endpoint⁸), asking for particular entities. Hence, there is no need to precisely know externally-defined entities, for instance, it is not required to know the DBpedia identifier for a country, some of which cannot be guessed trivially (e.g., http://dbpedia.org/page/Georgia_(country)). Querying external SPARQL endpoints is transparent to the user. A list of all currently existing countries is retrieved and cached by running a query against the DBpedia SPARQL endpoint. This list is periodically updated to obtain new updates, if exist. It is worth to mention that no special configurations for SPARQL endpoints are needed. Currently, SPARQL-AG is tailored to generate and execute queries over the EVENTSKG knowledge graph. This follows the motivation that the “the more a system is tailored to a domain, the better its retrieval performance is” [12]. However, the approach is easily transferable to other datasets and domain representations. Our research aims at answering the following questions:

- How can users query semantic data without knowing the schema of this data?

---

⁶ https://dbpedia.org/sparql
⁷ http://kddste.sda.tech/SEOontology/sparql
⁸ http://kddste.sda.tech/sparql
– How can users query semantic data without learning RDF, OWL, or SPARQL?
– How can we combine data from several SPARQL endpoints to formulate a
SPARQL query?

SPARQL-AG is a web-based user interface, which allows end users to create and
execute both simple and complex SPARQL queries over scholarly knowledge
bases. Generally, we believe that SPARQL-AG closes an important gap between
researchers outside the semantic web community, or even within the community
but not being SPARQL experts, and the semantic data available on the Web.
The service is publicly available online at http://kddste.sda.tech/SSR-Service/
SPARQL-AG/SPARQL-AG.php. It has been tested by several SPARQL experts
by creating a large number of successful queries. The source code is available on
GitHub (see Table 1).

The remainder of this article is organized as follows: Section 2 gives an over-
view on related work. Section 3 presents the design principles considered when
developing SPARQL-AG. Section 4 outlines the methodology we used. Section 5
presents a use case. Section 6 discusses the implementation and results of the
evaluation. Finally, Section 7 concludes with an outline of future work.

2 Related Work

The origins of query builders go back to preliminary research works in the
1990s [3]. Natural language interfaces (NLIs) are widely used to ease the process
of querying semantic data [14, 12, 16, 5]. Many contributions have been made
for this purpose; below, we present the state of the art. Most of these contribu-
tions use NLIs in two different ways: generating SPARQL queries based on
User Interactions (UI), and answering user queries using a Question Answering
system (QA). The latter completely hides SPARQL queries from the user, al-
lowing them to directly submit their question, e.g., NLP-Reduce [13] and Power-
Aqua [16], whereas the former focuses on generating SPARQL queries using a
visual interface, e.g., Semantic Crystal [12], Querix [14], and SPARQL Views [5].

UI-based systems. NLI-based systems are often tailored to a specific applica-
tion and require exceptional design and implementation efforts. Below we present
some of the state-of-the-art efforts in SPARQL query building using NLIs.
Semantic Crystal [12] is a graphically-based query tool that can be used for query-
ing OWL knowledge bases by generating SPARQL queries. The generated query
is composed by clicking on ontology elements from the ontology graph displayed
on a screen and selecting elements from menus. Querix [14] is an NLI-based
tool that translates natural language questions, written in English, to SPARQL
queries with little user interactions. One drawback of Querix is that it does not
resolve ambiguities in the input text, but asks the user for clarification. SPARQL
Views [5] is an NLI-based tool that supports visual query building via drag and
drop over RDF data in a Drupal CMS⁹. Via an auto-complete search box, users
can filter predicates, which can be used in the query pattern. QUaTRO2 [1]

⁹ https://www.drupal.org/
provides a graphical user interface to formulate complex queries based on an abstract domain-driven query language. QUaTRO2 tool has been used to query the UniProt\(^{10}\) protein database. QueryVOWL \(^{11}\) is a visual query language tool for creating SPARQL queries using GUI controls. This tool is developed based upon the VOWL \(^{15}\) ontology visualization.

In contrast to much of the existing work on building SPARQL queries, which tends to focus on translating natural language queries to SPARQL, which is still far from efficient, SPARQL-AG completely uses graphical interface controls to generate SPARQL queries, requiring prior knowledge neither about the schema being queried nor the SPARQL syntax. On the other hand, one of the limitations of our previous work on analyzing scholarly data \(^{8,9}\) is that the analysis was based on predefined queries. These analyses cannot be flexibly extended by changing the parameters of these queries. Therefore, more work is needed regarding the use of NLIs for facilitating the process of querying distributed semantic data, for both end users and SPARQL experts, and further comprehensive usability studies to investigate the end users’ perspective are required.

3 Design Principles

Design principles involve principles for: system design, portability, availability, maintainability and sustainability, and documentation.

**System design.** When designing SPARQL-AG, we integrate different research paradigms in the system architecture, illustrated in Figure 1: 1) query building (QB) – to build error-free SPARQL queries based on user selections from a visual interface. QBs have the advantage of allowing for high expressivity while assisting users by listing eligible query elements without prior knowledge about the syntax of the language. This helps to completely avoid syntax errors, 2) semantic search (SS) – for entities in various knowledge graphs that match the query pattern, 3) Human Computer Interaction (HCI) – to make the user’s interaction as simple and efficient as possible, in terms of accomplishing user goals, i.e., facilitating the task of querying semantic data without writing any piece of code, and 4) Federated Query – SPARQL 1.1 Federated Query is a technique that is used for executing queries distributed over different SPARQL endpoints.

**Portability.** To promote portability, SPARQL-AG is a fully web-based service following web standards. A public SPARQL endpoints is used for querying knowledge graphs using HTTP requests, PHP and JavaScript for the application code, and HTML5/CSS3 (Bootstrap) for designing and styling the user interface (more details in section 6).

**Availability.** SPARQL-AG has been available online at (http://kddste.sda. tech/ER-Service/SPARQL-AG/test/SPARQL-AG.php) since December 2018. Users only need the URL of the service to be able to use it, i.e., no configuration or prerequisites are required.

**Maintainability and Sustainability.** SPARQL-AG is developed and maintained by the first author and hosted in our server. To ensure the sustainability of

\(^{10}\) https://www.uniprot.org/
Table 1: SPARQL-AG-related resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENTSKG</td>
<td><a href="http://kddate.sda.tec/VENLESS-KG-Dataset/EVENTSKG_A2.html">http://kddate.sda.tec/VENLESS-KG-Dataset/EVENTSKG_A2.html</a></td>
</tr>
<tr>
<td>SEO Ontology</td>
<td><a href="http://purl.org/seo/">http://purl.org/seo/</a></td>
</tr>
<tr>
<td>SPARQL-AG URL</td>
<td><a href="http://kddate.sda.tec/SEM-Service/SPARQL-AG/SPARQL-AG.php">http://kddate.sda.tec/SEM-Service/SPARQL-AG/SPARQL-AG.php</a></td>
</tr>
<tr>
<td>GitHub repository</td>
<td><a href="https://github.com/saidfathalla/SPARQL-AG">https://github.com/saidfathalla/SPARQL-AG</a></td>
</tr>
<tr>
<td>Issue Tracker</td>
<td><a href="https://github.com/saidfathalla/SPARQL-AG/issues">https://github.com/saidfathalla/SPARQL-AG/issues</a></td>
</tr>
</tbody>
</table>

SPARQL-AG, we use the issue tracker on its GitHub repository (cf. Table 1) in order to make it easier for users to request new features, e.g., features not covered in the initial release, and to report any problems/bugs.

Documentation. The documentation of how to use SPARQL-AG and a set of pre-defined queries are available through its URL.

4 Architecture

The architecture of SPARQL-AG is composed of five components (see Figure 1): user interface, components selection, query composer, SPARQL clients manager, and query executor. Later in this section, we denote the set of SPARQL variables with $V$, the set of predicates used in the query pattern with $P$, the set of RDF resources with $R$, the set of query restrictions with $QR$, and the set of RDF literals with $L$.

User Interface. The role of the user interface component is to 1) provide end-users a Graphical User Interface (GUI) through which they can select different query components using a web form, 2) display the generated query (GQ) to the user, who can then modify it before it is submitted to the SPARQL endpoint for execution, and 3) submit the queries to the EVENTSKG SPARQL endpoint and display the results in a human-readable format (HTML table). SPARQL-AG features are: 1) generating and executing simple SPARQL queries, 2) generating SPARQL queries with aggregation, and 3) executing predefined query templates. Users are able to 1) select columns they want to appear in the result using checkboxes, 2) restrict the results by selecting the checkbox corresponding to each predicate and by entering/selecting possible object values for these predicates by depending on the datatype direct input of numeric values, selecting from a list, or picking a date from a calendar. This avoids the problem of resolving the ambiguity that might arise when processing natural language queries and irrelevant queries.

Components Selections. User selections of the different SPARQL query components, listed below, are bypassed to the Query Composer (QC) in order to formulate the SPARQL query based on these selections. 1) Prefixed declarations: the set of namespace prefixes used in the query, 2) Dataset definition: we omit the dataset definition part of the SPARQL query from user selections, because it is implicitly given, i.e., the EVENTSKG URI, 3) Result clause: identifying what information to return from the query, 4) Query patterns: specify the query’s graph
pattern that matches the data, such as UNION, MINUS, FILTER, and OPTIONAL, and 5) **Query modifiers**: a set of modifiers for the query results, such as order, projection, distinct, offset, and limit. Each selection is mapped to a query element in the generated query. To address scalability issues, users can limit the number of results retrieved using the LIMIT modifier.

**Query composer.** The query composer is the core component of SPARQL-AG, as it formulates the SPARQL query based on user selections and data received from the SPARQL Clients Manager, which collects data from external SPARQL endpoints. The main steps carried out by the Query Composer are summarized in algorithm 1, where "+" denotes string concatenation. First, users should define the namespaces used in the query in the prefix set \(P\). Currently, as SPARQL-AG is provided for the EVENTSKG dataset, the prefix declaration is automatically generated with the required predicates, i.e., **seo** and **conference-ontology**. For future purposes, if a namespace mapping is not available in the system, then the user should add this namespace to the prefix declaration. Each selection in the result clause (RC) (upper part in Figure 2) is mapped to a new variable in the SPARQL query (call mapResultClause). For instance, when **country** is selected, it represented by the variable ?country. Formally, the mapping is defined as follows: Each RC selection is assigned a unique variable via the function mapResultClause : RC → V. Each query pattern is represented as a tuple of \((prop, op, val)\), where prop is the property being restricted, op is the operation, and \(val \in L\) is the value. For example, \((\text{acceptanceRate}, \geq, 0.25)\) represents the
results with an acceptance rate greater than or equal to 0.25. Query pattern might contain externally defined entities, such as countries and cities (defined in DBpedia), and the research field which the events belong to (defined in SEO ontology). Here, the External Entities Table (EET) plays its role, in which all external entities are stored along with the URL of the public SPARQL endpoint of the knowledge graph in which these entities can be found. Therefore, these entities should be identified (call isExternallyDefined). The function isExternallyDefined : P → B is defined as: isExternallyDefined(p) := true if p is found in EET, and false otherwise. The URIs of these external entities should be retrieved as well by sending requests to the SPARQL Clients Manager via the function: mapExternalEntity : P → R. Each request is assigned a unique number via the function reqID : RR → N, where RR is the set of requests made by the Query Composer component. After successful retrieval of the requested data, all these requests are stored in the requests table (RT) for answering further requests, instead of sending them to the Clients Manager, i.e., caching requests. Therefore, after a period of time, when all external entities have been requested, there is no more need to query external endpoints for these entities. This reduces the workload of querying external knowledge graphs for every request. In addition, it performs results aggregations when more than one client returns a result.
Algorithm 1: QueryGeneration

query = null;
foreach namespace nt ∈ PS do
  query += "PREFIX " + nt + ";
query += "SELECT ";
while rc is not empty do
  // add result clause elements query += "?" + mapResultClause(rc);
query += "WHERE ( ";
foreach pattern Pi ∈ QP do
  if uriisaExternallyDefined(Pi) then
    value = mapExternalEntity(Pi);
  else
    value = UI control.value;
  // to get literal values, e.g. numeric values from the UI
  query += " ?e " + mapToPredicate(Pi, prop) + mapToVariable(Pi, prop) + " FILTER ( mapToVariable(Pi, prop) = Pi, op = Pi, val = " + value + ")";
query += ");"

In order to map query restriction (QR) to the corresponding predicate in the dataset, the function mapToPredicate : QR → P is used. For instance, when the user wants to filter results by, e.g., country, then he/she should select the country checkbox (Figure 2), which should be mapped to the corresponding predicate in the dataset, i.e., seo:heldInCountry. Each variable in the result clause must be bound in the query pattern, therefore the function mapToVariable : P → V is used to obtain these variables, which are defined in the result clause to be bound in the query pattern. For example, when users want to display events along with their start and end dates, these attributes are bound to two variables in the result clause, which are ?SD and ?ED respectively. Mapping query modifiers is straightforward, e.g., the order by modifier specifies either one column or two columns to order the results by, after the ORDER BY keyword.

Query Validator. This component is responsible for validating the generated query before sending it to the SPARQL endpoint for execution. In addition, when users modify the generated query before execution, they can validate the modified query using the "Validate" button. SPARQL Clients Manager. This component is responsible for managing SPARQL clients in order to be able to query external SPARQL endpoints, DBpedia, and SEO in this case. SPARQL clients allow executing SPARQL queries against remote SPARQL endpoints using the SPARQL protocol [10]. After a successful connection to the endpoint, the SPARQL client sends the SPARQL query to the endpoint and waits for the result. When requests are received from the Query Composer, asking for the resource URI of an externally defined entity, e.g., Germany, the Clients Manager formulates a SPARQL query and sends it to the SPARQL client responsible for this type of request and waits for the requested resource URI, e.g., (http://dbpedia.org/resource/Germany).
Query Executor. This component is responsible for sending the validated query to the Clients Manager (AskForResults) and displaying the results in a human-readable format. Since the query results are returned as an array of variable bindings, which are difficult to understand for end-users who are not familiar with SPARQL, we decided to display the results as an HTML table.

Generating SPARQL with aggregation functions. Aggregation functions are useful when users want to study data in an analytical fashion, e.g., finding the total number of publications of all events in each research field. Results are grouped using the GROUP BY clause, and these groups can be filtered using the HAVING clause. Here, it is worth to mention that SPARQL-AG enforces some SPARQL rules to be applied while users select different query elements. For instance, a group column is added to the GROUP BY clause when aggregation functions are used, and aggregation functions are automatically restricted using HAVING. For example, when the user uses a column in an aggregation function, this column must be added to the GROUP BY clause; this is a SPARQL restriction.

5 Running Example and Use cases

In this section, we present use cases for SPARQL-AG for supporting scholarly communication stakeholders by providing figures about computer science events in the context of eight computer science communities. Listing 1 shows the SPARQL query generated for the query “List the top-10 events with topics related to Artificial Intelligence with an acceptance rate lower or equal to 0.20, which have been held in Germany in the last decade. Order the results ascending by acceptance rate”. Listing 2 shows the SPARQL query generated for a query with an aggregation: “List the subfields of computer science whose events have a large number of submissions, i.e., greater than 10,000. Order the results by ascending field name”.

Listing 1: SPARQL query generated based on user selections in Figure 2.

```sparql
PREFIX seo: <http://purl.org/seo/>
PREFIX conference-ontology: <https://w3id.org/scholarlydata/ontology/conference-ontology.owl#>
SELECT DISTINCT ?event ?acceptance ?field WHERE {
  ?event seo:heldInCountry ?country .
  FILTER (?country = <http://dbpedia.org/resource/Germany>) .
  ?event seo:field ?field .
  FILTER (?field=<http://purl.org/seo#ArtificialIntelligence>) .
  ?event seo:acceptanceRate ?acceptance .
  FILTER (?acceptance <0.20) .
} ORDER BY ?acceptance LIMIT 10
```

Listing 2: SPARQL query generated based on user selections in Figure 2.
6 Implementation and Evaluation

This section describes the implementation, the evaluation and discusses the results of a usability study for testing SPARQL-AG.

Implementation. SPARQL-AG is implemented in PHP as a web-based service, using a client-server architecture. We have implemented all functions described in section 4 using PHP 7.2.10 and the RAP11 (RDF API for PIIP) toolkit. In addition, JavaScript is used to validate input data and to enforce some rules, such as GROUP BY rules, as mentioned in section 4. SPARQL-AG only needs the URL of an endpoint to explore it, without any further required configuration. Thus the approach is easily transferable to other datasets by just changing the components selections and the dataset URL. Queries to SPARQL endpoints are sent directly from the client browser, using HTTP requests, which makes SPARQL-AG independent from a server. Usability testing is a technique used in user-centered interaction design to evaluate a product by testing it by letting real users use the system [17]. Nielsen and Landauer [17] argue, that the best usability evaluation results come from testing no more than five users and running as many small tests as possible. This type of evaluation was performed by several Semantic Web query interfaces [12, 5] since it gives direct insight into how real users use the system. The goal is to improve the usability of the system being tested.

Experimental Setup. In this experiment, casual end-users should test and assess the usability of the service. To apply the usability test for SPARQL-AG with real-world end-users, we promoted the usability study on its web site, several mailing lists and between colleagues. A total of ten participants were recruited for this study. They are distributed over a wide range of backgrounds and professions. In addition, we ensure the anonymity of the participants in order to obtain unbiased results. The participants were split down into two groups based on their SPARQL experience: 1) experienced SPARQL users (all are computer scientists), and 2) casual end-users from other fields and professions. To assure the usefulness of SPARQL-AG, we confirm that it is able to answer a number of competency queries listed in Table 2. These queries were thoroughly selected to assure that they cover all features provided by SPARQL-AG. In the beginning, we informed all participants that the query interfaces were being tested and not

SELECT ?field SUM(?submissions) AS ?SP_SUM WHERE {
  ?event neo:field ?field .
}
GROUP BY ?field
HAVING (SUM(?submissions)>10000)
ORDER BY ?field
LIMIT 10

11 http://wifo6-03.informatik.uni-mannheim.de/bizer/rdfapi/index.html
Table 2: Queries used in evaluating SPARQL-AG. Each variable, such as X and Y, is a placeholder for any appropriate replacement.

<table>
<thead>
<tr>
<th>No.</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>List events related to field X that took place in country X.</td>
</tr>
<tr>
<td>Q2</td>
<td>List events related to field X with an acceptance rate less than Y along with their sponsors, publishers, website and start date.</td>
</tr>
<tr>
<td>Q3</td>
<td>List events that took place in a particular month X along with their publishers and the field of research.</td>
</tr>
<tr>
<td>Q4</td>
<td>List conference series that have been held in country X in a particular month Y.</td>
</tr>
<tr>
<td>Q5</td>
<td>List the number of submitted and accepted papers of a series X in a particular time period.</td>
</tr>
<tr>
<td>Q6</td>
<td>List the top-X countries hosting most of the events in computer science research communities.</td>
</tr>
<tr>
<td>Q7</td>
<td>List the subfields of CS for which a country X has hosted most events since a particular date Y.</td>
</tr>
<tr>
<td>Q8</td>
<td>Compare the popularity of different computer science research communities, in terms of the number of submissions to the respective events.</td>
</tr>
<tr>
<td>Q9</td>
<td>List the top-X research fields, in terms of the number of events they have.</td>
</tr>
<tr>
<td>Q10</td>
<td>Find the average acceptance rate for events in each computer science research community.</td>
</tr>
</tbody>
</table>

the users themselves. This is an important issue that can severely influence the test results. Inexperienced users are confused when given too many interaction options. Therefore, at the beginning of each experimental run, we gave each participant all information and instructions concerning the experiment either in a face-to-face meeting or in a call (for remote participants). Most of the users found that the experiment can be easily understood by casual end-users and does not require expert knowledge. After testing the service, experienced users were explicitly asked to fill in a satisfaction questionnaire (can be accessed through the link https://goo.gl/55T8R0) in which they were asked about their assessment of the interface, generated query, the presentation of the results, and the usefulness of SPARQL-AG for both casual and experienced users. In addition, casual users were explicitly asked to fill in the System Usability Scale (SUS) questionnaire [4] (available at https://goo.gl/4xj9Ua). SUS is a standardized usability test, which contains ten questions with five possible responses ranging from 1 (strongly disagree) to 5 (strongly agree). The best way to interpret one’s results is by normalizing the scores to produce a percentile ranking. We also asked users for further qualitative feedback, including positive and negative aspects, and suggestions for future improvements.

Results. For the analysis of results, the SUS scoring method [18] has been used for the the casual end user questionnaire. The average SUS score falls into seven adjective ratings, ranging from Best Imaginable (above 90.9) to Worst Imaginable (below 12.5) [2]. Most strikingly, findings showed that SPARQL-AG scored a high SUS satisfaction score of 93, i.e., Best Imaginable, thus reach-
Table 3: Statistics for questions of the SUS Questionnaires for casual end-users.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>4.57</td>
<td>4.14</td>
<td>4.71</td>
<td>4.57</td>
<td>4.57</td>
<td>4.00</td>
<td>4.86</td>
<td>4.43</td>
<td>4.29</td>
<td>3.86</td>
</tr>
<tr>
<td>SD</td>
<td>0.53</td>
<td>1.46</td>
<td>0.49</td>
<td>0.53</td>
<td>0.53</td>
<td>1.00</td>
<td>0.38</td>
<td>1.13</td>
<td>0.76</td>
<td>0.90</td>
</tr>
<tr>
<td>Median</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
<td>4.00</td>
<td>5.00</td>
<td>4.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Mod</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
<td>4.00</td>
<td>5.00</td>
<td>5.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>

Figure 3: The mean of experts response in the satisfaction questionnaire.

ing excellent usability. Table 3 contains the statistics for each question of the SUS Questionnaires for casual end-users. Since all questions measure positive agreement, notably, the mode of almost every response is 5, which means that most participants responded with strong agree for most questions. As shown in Figure 3, the mean of expert responses to the expert questionnaire falls into the range of 3.24.8, which is rather high. This implies a fair satisfaction of all experts for all questions, since all questions measure positive satisfaction.

7 Conclusions and Future work

This work discusses SPARQL-AG, which aims at improving data access to semantic knowledge graphs by generating SPARQL queries for users who may be challenged by lack of schema or data knowledge regarding a knowledge graph. We present the architecture of SPARQL-AG, which can be used for generating SPARQL queries for any semantic data. These components combine research techniques from different disciplines in an integrated fashion. We highlight the importance of using NLIs to make semantic data accessible to a wider community. One aim of this study is to show the potentials of NLIs that give a chance to casual users to benefit from the Semantic Webs capabilities without having to study them. In fact, SPARQL-AG also significantly lowers the barrier of writing SPARQL queries from scratch, by providing additional support for SPARQL experts. It generates SPARQL queries of three kinds: simple query
generation, SPARQL query generation with aggregation, and parametrized execution of predefined queries. We believe that we are on the way towards increasing users understanding of SPARQL by lowering the syntax barrier, since generating queries using user interactions will increase the understanding of the syntax itself, enabling users to incrementally improve their understanding of the query language. Our usability study with ten participants using a list of 21 different questions showed that the usability of the system is excellent, with a SUS score of 93. The results of our evaluation show that both experienced and casual users agree that writing SPARQL queries in a blank sheet, where they must type commands, is cumbersome and time-consuming. As anticipated, SPARQL-AG enables successful generation of error-free and readable queries which potentially saves much time and effort. To the best of our knowledge, this is the first web-based user interface, which allows end users to create and execute both simple and complex SPARQL queries over scholarly knowledge bases. Our work has some limitations. Still not all SPARQL 1.1 specifications are covered. Also, currently, it is restricted to only one dataset. Nevertheless, we our work provides a framework for developing query builders for querying scholarly data, making such data available for further analysis and improvement.

There are several directions for future work along the three dimensions: 1) Extension: we are planning to extend SPARQL-AG’s functionality, especially with respect to expressiveness, and interface robustness. For expressiveness, we plan to cover almost all SPARQL 1.1 features, in particular, subqueries, multi-dimensional queries, nested aggregations for rich analytics, and graphs as results for CONSTRUCT queries and updates. For interface robustness, we are planning to adopt the interface to let the user select the knowledge base to be queried, and improve the interface based on participants feedback. 2) Evaluation: Furthermore, a more comprehensive evaluation of the services will be done after the implementation of the new features by considering more complex queries in different domains. 3) Data visualization: we are planning to display results using graphical visualizations, e.g., charts and graphs.

References

REFERENCES