Investigating Learnability, User Performance, and Preferences of the Path Query Language SemwidgQL Compared to SPARQL

Timo Stegemann(✉) and Jürgen Ziegler

University of Duisburg-Essen, Duisburg, Germany
timo.stegemann@uni-due.de
http://interactivesystems.info

Abstract. In this paper, we present an empirical comparison of user performance and perceived usability for SPARQL versus SemwidgQL, a path-oriented RDF query language. We developed SemwidgQL to facilitate the formulation of RDF queries and to enable non-specialist developers and web authors to integrate Linked Data and other semantic data sources into standard web applications. We performed a user study in which participants wrote a set of queries in both languages. We measured both objective performance as well as subjective responses to a set of questionnaire items. Results indicate that SemwidgQL is easier to learn, more efficient, and preferred by learners. To assess the applicability of SemwidgQL in real applications, we analyzed its expressiveness based on a large corpus of observed SPARQL queries, showing that the language covers more than 90% of the typical queries performed on Linked Data.

1 Introduction

The wealth of Linked Data published on the open Web [15] offers a wide range of opportunities that are to date still underexploited in practical applications. Integrating Linked Data from different sources into standard web sites, blogs or other web applications would enable web authors and developers to reuse the vast amount of information already available and create additional value by enriching their content or by syndicating different data sources. However, a more widespread use of textual and multimedia resources from Linked Data and, even more so, of time-dependent data from the Internet of Things is currently significantly hindered by their complexity. It thus seems important to lower the threshold for users such as web developers or even normal web authors by providing techniques for using Linked Data without requiring complicated technical installations or the knowledge of powerful yet complex query languages such as SPARQL.

To alleviate the problems involved in using linked data, we have developed a JavaScript-based environment, that facilitates the integration of Linked Data in web pages. A main component of this environment is the path query language SemwidgQL that is intended to be significantly easier to use than standard SPARQL. A first overview of the SemwidgJS environment was presented

C. d'Amato et al. (Eds.): ISWC 2017, Part I, LNCS 10587, pp. 611–627, 2017.
DOI: 10.1007/978-3-319-68288-4_36

in [16]. In this current paper, we focus on the path query language developed and provide a description of its novel extensions. We further present a comprehensive empirical user study comparing SPARQL and SemwidgQL. The goal of this study is to explore SemwidgQL's effectiveness, efficiency, learnability for users, and user preference in comparison to SPARQL. The study supports our claim that SemwidgQL is easier to learn, more efficient, and preferred by the learners. To investigate how well SemwidgQL covers the range of SPARQL queries used in practice, we further analyzed several hundred thousand log entries of public SPARQL endpoints. Results indicate that SemwidgQL can cover most of the requests that are currently made with SPARQL.

2 Related Work

Several approaches to support querying, exploring and displaying Linked Open Data have been described in literature so far. Many of these approaches are specialized on browsing (e.g. [2,7]) and visualizing queried data values or sub graphs (e.g. [8]) respectively, which is generating revealing insights but can only be reused on other websites with large effort.

FSL [11] and LDPath [14] are path query languages for RDF data inspired by XPath for XML. LDPath is part of the Apache Marmotta platform for Linked Data. One drawback of these languages is that they can only return single result lists but not lists of results sets, which is necessary when querying a set of different properties at once. Requesting coherent values from different properties require distinct queries that request each property separately. Therefore, it is not provided that these values stay connected, since the respective order can be different, or values can be added to or removed from the data set between requests. Rules for a translation into SPARQL do not exist for these languages and therefore they require direct access to the data or a special interface on the server side. Language extensions for SPARQL such as C-SPARQL [1] and SPARQLStream [5] facilitate the usage of queries over streams of RDF data. Time windows that restrict the queried data to a period of time can be specified in a special FROM STREAM statement.

The performance of users for different query languages has already been evaluated in pre-SQL times [12]. However, since the effort is very high, user studies that compare different query languages are rarely conducted. Participants require an extensive introduction to be able to use a query language at a satisfying level. Mostly this happens in the context of a lecture. We are not aware of any user study that compares the participants' performance with SPARQL and another query language of the Linked Data area.

3 SemwidgQL

SemwidgQL is a path query language that transcompiles to SPARQL. In contrast to queries formulated in other Linked Data path query languages, such as LDPath [14] or FSL [11], SemwidgQL can therefore be used to query any public

SPARQL endpoint without further special requirements. Unlike these languages, SemwidgQL is also capable of querying sets of different properties at once, by adding all properties of interest to the SELECT statement of its SPARQL translation. Ultimately, SemwidgQL aims to combine the benefits of SPARQL (such as its prevalence in the Linked Open Data area or its ability of returning lists of result sets) with the simplicity of path query languages.

In the following section, we give an overview of SemwidgQL's core features that have been described in more detail in a previous publication [16]. SemwidgQL has been significantly extended since then and we present further features that were added to facilitate among others the querying of time-sequential data, such as sensor data, and give experienced users more control over the generated SPARQL queries via filters and pseudo-filters. Specifications of time windows are comparable to the approaches of SPARQL streaming extensions. In contrast to these extensions, SemwidgQL is compatible with regular SPARQL endpoints and requires no additional execution environment.

3.1 Core Features

As a path query language, SemwidgQL traverses RDF graphs. The traversal is indicated by the dot notation, which is reminiscent of the well-known syntax used in object-oriented programming. Figure 1 shows the simplified basic structure of a SemwidgQL query. Usually a query starts with a resource followed by one or more properties. To further filter the result set, properties can be restricted. Filters are enclosed in parentheses and are appended the property they restrict. The left-hand side of a filter expression is typically a property (or a property path) that refers to the property to restrict outside of the parentheses. The right-hand side specifies a filter value that can be a literal, IRI, or even a nested query. Between them stands a relational operator. Several filter expressions can

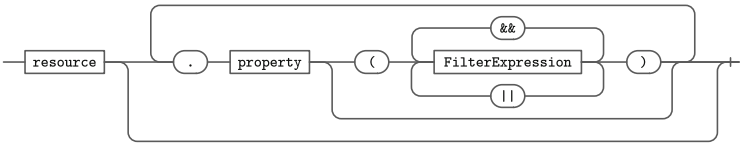

Fig. 1. Basic structure of a SemwidgQL query.

Table 1. Basic SemwidgQL queries and their meanings.

SemwidgQL query	Technique	Meaning
`dbr:Vienna.rdfs:comment`	Path Expression	Textual description of Vienna
`dbr:Vienna.^dbo:capital`	Inverse Property	Country, where Vienna is the capital
`dbr:Vienna.^dbo:birthPlace` ↪ `(rdf:type = dbo:SoccerPlayer)`	Filter	Soccer players, who were born in Vienna

be combined by logical operators. Furthermore, SemwidgQL allows wildcard selectors, inverse property selections, and multiple property selections. Table 1 shows some exemplary SemwidgQL queries.

3.2 Advanced Features

In addition to SemwidgQL's core features, we have implemented several filter and pseudo-filter keywords that, among others, simplify restricting language of string literals or allow aggregation of results. Also, they facilitate querying of time-sequential data with flexibly specified sampling intervals. Filter and pseudo-filter keyword expressions can be combined with normal SemwidgQL filter expressions and with each other as well. While filter expressions in SemwidgQL result in filter expressions in SPARQL, pseudo-filter expressions can have an impact on different parts of the translated query. An overview of these expressions is given below.

Filter Expressions

`@lang:` With this keyword the language of the property can be filtered by the given language code.
`@self:` This keyword refers to the property to restrict itself. Instead of filtering a property that is related to the property to restrict, it can be filtered directly.
`@timestart`/`@timeend:` These keywords allow the filtering of values after, before, or (when combined) between two points of time. The right-hand side of the expression can be an absolute date or a relative point in time, depending on the time of the query execution. The expression is parsed as an equation, whose first part is a timestamp or the term `now` followed by the amount of time that has to be added or subtracted. This can be expressed in seconds, minutes, hours, day, weeks, or a combination of these (e.g. `now - 1h 5min`).
`@type:` This keyword is equivalent to the property `rdf:type`.

Pseudo-Filter Expressions

`@aggregate:` This keyword allows to apply an aggregate function to the variable of the property within the SELECT statement. Allowed values are COUNT, SUM, MIN, MAX, AVG, and SAMPLE.
`@hide:` If set to `true`, the variable of the property will not be part of the SELECT statement.
`@optional:` If set to `true`, the triple pattern, in which the property is created, will be enclosed in an OPTIONAL statement.
`@predicate:` Typically, the predicate of a triple pattern is not part of the SELECT statement. If set to `true`, the predicate of the triple pattern, in which the property is created, will be added to the SELECT statement.
`@timeinterval:` This keyword is used to group and aggregate time-sequential values. On the right-hand side of the expression, a sampling interval can be

defined. All returned values within this interval will be aggregated. By default, the SAMPLE aggregate function will be applied to all variables, but different functions can be specified by the `@aggregate` keyword. Similar to `@timestart` and `@timeend`, the length of the interval can be expressed in seconds, minutes, hours, day, weeks, or a combination of these.

```
*(ip:sensor = ir:TH_LF285 && ip:type = 'Temperature').[ip:value(@aggregate = 'AVG'),
ip:measuredAt(@timeinterval = '60 min' && @timestart = 'now - 7 days' && @aggregate = 'MIN')]
```

⇓

```
SELECT DISTINCT SAMPLE(?wildcard) AS ?wildcard
    AVG(?value) AS ?value MIN(?measuredAt) AS ?measuredAt
WHERE {
  ?wildcard ip:value ?value .
  ?wildcard ip:measuredAt ?measuredAt .
  ?wildcard ip:sensor ?sensor .
  ?wildcard ip:type ?type .
  FILTER (
    ?sensor = ir:TH_LF285_01 && STR(?type) = "Temperature"
  )
  FILTER (
    xsd:dateTime(?measuredAt) >= now() - 604800
  )
  BIND(FLOOR((xsd:dateTime(?measuredAt) -
      xsd:dateTime("1970-01-01T00:00:00")) / (3600)
      ) AS ?measuredAt_timeinterval)
}
GROUP BY ?measuredAt_timeinterval
ORDER BY DESC(?measuredAt_timeinterval)
```

Fig. 2. A SemwidgQL query and the corresponding SPARQL query that requests the average temperature measurements that were made by a specific sensor during the last week, aggregated on an hourly base.

A SemwidgQL query and its rather complex translation into SPARQL is shown in Fig. 2. The query contains normal SemwidgQL filter expressions combined with previously presented filter and pseudo-filter keyword expressions. It requests the average temperature measurements that were made by a specific sensor (located in our office) during the last week, aggregated on an hourly base.

4 Empirical User Study

We conducted an empirical user study comparing SPARQL and SemwidgQL. SemwidgQL was developed to be effective, efficient, and easy to learn by non-expert users. The goal of our study is to explore whether SemwidgQL can fulfill these requirements in comparison to SPARQL. In addition we want to investigate the users' satisfaction. At the beginning of this section, we will describe the design of the study and its procedure. Afterwards, we will present the results. In conclusion, we interpret and discuss these results.

4.1 Method

Design: We conducted an empirical user study with a mixed methods design and repeated measures, combining objective performance measures and a subjective questionnaire. For the performance measure, participants had to complete several query interpretation and formulation tasks. Effectiveness was measured by the number of correct answers of all query tasks.

Efficiency was measured by the participants' performance measures for the query formulation tasks. We investigated these tasks regarding nine dependent variables, i.e. (a) *number of keystrokes* (number of keystrokes made by a participant, including deletion and substitution of characters), (b) *number of corrections* (number of correcting keystrokes made by a participant, such as backspace, delete, replacing several selected characters etc.), (c) *number of conjunct corrections* (a coherent sequence of correcting keystrokes forms a conjunct correction, e.g. multiple backspaces in succession; typing of a character ends a conjunct correction), (d) *number of pauses* (number of pauses taken by a participant; a pause starts after two seconds without a keystroke; a pause might be an indicator for that a participant requires some time to think about further actions that are required to solve the task), (e) *time of pauses* (accumulated time of pauses in seconds taken by a participant during a task; operationalizes thinking time), (f) *time on task* (processing time of a task in seconds), (g) *number of requests* (number of requests a participant made to the SPARQL endpoint), (h) *fraction of erroneous requests* (fraction of requests that could not be executed due to parser errors etc.), (i) *display time of solutions* (time in seconds that a participant inspected the sample solution; a high display time might be an indicator that participants are uncertain about their solutions and therefore compare their own and the model solution more thoroughly).

Learnability was evaluated by comparing the results of the query formulation tasks from the initial and repeated measures regarding the above listed variables.

User preferences were measured through the answers from the questionnaire. The questionnaire asked to rate six characteristics of SPARQL and SemwidgQL on the basis of an equidistant five-point numerical rating scale. The minimum value always had a negative and the maximum value always had a positive connotation. These items were related to the subjective assessment of SPARQL's and SemwidgQL's learnability, intuitiveness, logical structure, comprehensibility, writing effort, and sophistication. Also, the participants were explicitly asked for their personally preferred language and a brief explanation for their decision.

Participants: The study was attended by seven students (one female), all enrolled on master courses in computer science at our University. The age of the participants was between 23 and 28 years ($M = 25.57$; $SD = 2.07$). Three participants had already gathered previous experience in Linked Data and Semantic Web from different courses, and one student had already worked with RDF and SPARQL as part of his bachelor thesis.

Procedure: The user study took place in the context of the introductory session of a seminar on "Semantic Web Technologies and Applications" for graduate students in the field of computer science. At the beginning of the seminar, the participants were handed a three-page handout[1], which contained an overview of relevant SPARQL and SemwidgQL commands, as well as a small RDF graph that was used for all examples and tasks of the presentation and evaluation. The graph contained, among other things, some information about cities in the region, such as label, population, districts, class, but also temperature measurements of sensors. The data were chosen in such a way that the participants could compensate for misunderstandings through their personal context knowledge.

The introductory session consisted of a three-hour lecture which was divided into three one-hour sections. In the first section, the participants were taught the basic ideas, techniques and formats on which Linked Data and the Semantic Web are built. In the second section, the participants were given an introduction to SPARQL and in the third section an introduction to SemwidgQL. As far as it was possible, the procedure corresponded to the procedure of the previous section. Care was taken to explain both languages to a similar extent and it was ensured that the participants understood both languages at a comparable level.

Afterwards, the participants had to complete a set of twelve query tasks. In the first three tasks, they had to interpret predefined SPARQL and SemwidgQL queries. In the following nine tasks, they had to query predetermined information using SPARQL and SemwidgQL. Each task had to be processed with both languages. Namespace definitions were predefined for both languages. The order of the query languages changed at each task. At any time, participants could query the SPARQL endpoint and validate their queries and results. They could quit tasks at any time and move on to the next one. No time limit was set for solving a task. After each task, a model solution was presented.

Subsequently, the participants filled out a questionnaire in which they should specify socio-demographic information and previous experiences with Semantic Web and Linked Data techniques. Then they evaluated SPARQL and SemwidgQL regarding the above mentioned characteristics. One week after the introductory session, the study was repeated.

Data Collection: The data of the interpretation and formulation of queries were automatically collected via the specially prepared website on which the participants had to solve their tasks. Each keystroke was recorded and stored together with a time stamp in a central database. It was also recorded when the SPARQL endpoint was queried and it was recorded whether the query was valid or contained errors. The time stamps, at which the participants started or ended a task, the model solution was displayed, and a task was marked as successfully completed or marked as canceled by the participants, were recorded as well. The questionnaire data were collected via the online survey portal SoSci Survey[2].

[1] Handouts and tasks: https://semwidg.org/files/share/iswc2017_appendix.pdf.
[2] https://www.soscisurvey.de/.

4.2 Results

Correctness of Answers: Answers were divided into three categories. Correct answers, answers with minor errors, and incorrect answers. Answers with minor errors are syntactically correct and close to the model solutions, but can contain minor inaccuracies, such as queries that contain a triple pattern for requesting a desired property but do not contain the corresponding variable in the SELECT statement. Incorrect answers are syntactically incorrect, do not fulfill the requirements given in the task description, or the task was aborted by the user.

In total, we evaluated results of 147 tasks per language. From these results 21 belong to the query interpretation tasks and 126 belong to the query formulation tasks. The participants performed slightly better, when interpreting SemwidgQL queries compared to interpreting SPARQL tasks. Regarding SemwidgQL, 86% of the tasks were solved correctly, 14% of the solutions had minor errors. Regarding SPARQL, 76% of the tasks were solved correctly, and 24% of the solutions had minor errors. There were no incorrect answers in terms of the query interpretation tasks. With regard to the query formulation tasks, the participants achieved almost equally good results with both languages. Regarding SPARQL, 90% of the tasks were solved correctly, 6% of the solutions contained minor errors and 4% were incorrect. Regarding SemwidgQL, 89% of the tasks were solved correctly, 7% of the solutions contained minor errors and 4% were incorrect.

Query Formulation Tasks: In the following subsections, we will describe the results of the nine query formulation tasks (tasks 4–12), the participants had to solve during the evaluation. For each of the following statistical tests, we compared the participants' performance regarding the nine dependent variables listed in the study design subsection. For the subsequent tests, we restrict the examined data to pairs of correct answers or answers with minor errors, since data from incorrect or canceled solution would doubtlessly distort the results.

Analysis of Mean Performance: We compared the participants' performance regarding the above-mentioned dependent variables by calculating multiple dependent t-tests for paired samples. The further described results are presented in detail in Table 2. SemwidgQL's values regarding six of all nine dependent variables were significantly better compared to SPARQL. The *number of conjunct corrections (c)* was descriptively better regarding SemwidgQL compared to SPARQL. However, this difference is not statistically significant. The *number of requests (g)* and the *fraction of erroneous requests (h)* were better in SPARQL compared to SemwidgQL. These differences are also not statistically significant.

Analysis of Learning Effects: We evaluated, how the participants performance changed between the first and second pass of the user study. We also compared the differences between SPARQL and SemwidgQL during these two

Table 2. Differences between SPARQL and SemwidgQL.

	SPARQL		SemwidgQL		t-test		
	M	*SD*	*M*	*SD*	$t(124)$	p	
(a) number of keystrokes	136.32	58.90	74.56	55.02	14.33	<.001	***
(b) number of corrections	15.10	16.49	12.15	16.10	2.04	.044	*
(c) number of conjunct corrections	6.02	5.27	5.10	6.63	1.72	.088	
(d) number of pauses	11.06	7.95	7.57	7.36	5.82	<.001	***
(e) time of pauses (s)	99.82	122.89	67.84	73.91	2.99	.003	**
(f) time on task (s)	152.26	117.81	109.44	89.85	4.60	<.001	***
(g) number of requests	3.54	4.51	4.26	5.78	−1.45	.150	
(h) fraction of erroneous requests	0.19	0.27	0.21	0.27	−0.56	.578	
(i) display time of solutions (s)	7.56	10.75	4.67	4.60	2.83	.005	**

*$p < .05$, **$p < .01$, ***$p < .001$

passes. Again, we calculated multiple dependent t-tests for paired samples. Differences between SPARQL and SemwidgQL at each pass are presented in Table 3 and Fig. 3. Differences between the first and second pass for each language are shown in Table 4 and Fig. 3, in combination with the results of the previous tests.

In the first pass, results in terms of SemwidgQL were significantly better regarding four of the nine dependent variables compared to SPARQL, and descriptively but not significantly better regarding two further dependent variables. The participants never performed significantly better with SPARQL. In the second pass, results regarding SemwidgQL became significantly better at all but one dependent variable compared to the first pass. Results regarding SPARQL became significantly better regarding four dependent variables. All other results became descriptively but not significantly better. In comparison to SPARQL, participants performed significantly better with SemwidgQL regarding five of all nine dependent variables. Again, the participants never performed significantly better with SPARQL.

Complexity-Dependent Analysis: We compared the performance of the participants for SPARQL and SemwidgQL regarding a task's complexity. We assume that complexity of a task is a predictor for the measured responses. For this purpose, we conducted several linear regression analyses for the previously mentioned dependent variables and the complexity of a task as predictor variable.

In various works to determine the difficulty of SPARQL (e.g. [10]) or other (database) queries (e.g. [3,9]) Halstead's complexity measure [6] has been used. This measure is based on the number of distinct operators and operands as well as the total number of operands of a query or piece of source code. Halstead's complexity measure tends to produce comparatively high values when SPARQL queries contain filter expressions because the number of operators increases noticeably. Thus, it seems to overrate the influence of filter expressions on complexity. Because of this limitation, we developed an alternative complexity

Table 3. Differences between SPARQL and SemwidgQL per pass.

		SPARQL		SemwidgQL		t-test		
	Pass	*M*	*SD*	*M*	*SD*	$t(56)$	p	
(a) number of keystrokes	1	143.74	71.14	83.65	66.78	2.54	.014	*
	2	126.33	44.41	63.39	38.50	2.68	.010	**
(b) number of corrections	1	17.14	20.00	15.95	20.55	1.96	.056	
	2	12.28	11.28	7.74	8.81	3.40	.001	**
(c) number of conjunct corrections	1	6.09	6.05	6.44	8.82	0.71	.483	
	2	5.60	3.65	3.61	3.33	2.90	.005	**
(d) number of pauses	1	12.49	9.52	9.21	8.89	3.57	<.001	***
	2	8.79	4.87	5.54	5.03	4.00	<.001	***
(e) time of pauses (s)	1	102.98	120.26	79.85	84.21	1.03	.309	
	2	80.88	115.70	49.39	50.57	3.58	<.001	***
(f) time on task (s)	1	166.33	140.71	125.51	103.54	2.84	.006	**
	2	118.67	53.56	85.65	61.55	3.85	<.001	***
(g) number of requests	1	3.82	5.25	4.77	6.53	1.42	.161	
	2	2.89	2.66	3.11	3.18	2.53	.014	*
(h) fraction of erroneous requests	1	0.21	0.27	0.27	0.28	0.90	.374	
	2	0.17	0.26	0.12	0.22	3.77	<.001	***
(i) display time of solutions (s)	1	10.40	14.60	4.58	4.63	3.33	.002	**
	2	4.04	3.20	4.40	4.50	0.30	.765	

*$p < .05$, **$p < .01$, ***$p < .001$

Table 4. Differences between first and second pass per language.

		1st pass		2nd pass		t-test		
	Lang[a]	*M*	*SD*	*M*	*SD*	$t(56)$	p	
(a) number of keystrokes	$\mathcal{A}$	143.74	71.14	126.33	44.41	2.54	<.001	***
	$\mathcal{B}$	83.65	66.78	63.39	38.50	2.68	<.001	***
(b) number of corrections	$\mathcal{A}$	17.14	20.00	12.28	11.28	1.96	.641	
	$\mathcal{B}$	15.95	20.55	7.74	8.81	3.40	.004	**
(c) number of conjunct corrections	$\mathcal{A}$	6.09	6.05	6.44	3.65	0.71	.717	
	$\mathcal{B}$	6.44	8.82	3.61	3.33	2.90	<.001	***
(d) number of pauses	$\mathcal{A}$	12.49	9.52	8.79	4.87	3.57	.003	**
	$\mathcal{B}$	9.21	8.89	5.54	5.03	4.00	<.001	***
(e) time of pauses (s)	$\mathcal{A}$	102.98	120.26	80.88	115.70	1.03	.109	
	$\mathcal{B}$	79.85	84.21	49.39	50.57	3.58	.051	
(f) time on task (s)	$\mathcal{A}$	166.33	140.71	118.67	53.56	2.84	.012	*
	$\mathcal{B}$	125.51	103.54	85.65	61.55	3.85	<.001	***
(g) number of requests	$\mathcal{A}$	3.82	5.25	2.89	2.66	1.42	.264	
	$\mathcal{B}$	4.77	6.53	3.11	3.18	2.53	.650	
(h) fraction of erroneous requests	$\mathcal{A}$	0.21	0.27	0.17	0.26	0.90	.223	
	$\mathcal{B}$	0.27	0.28	0.12	0.22	3.77	.266	
(i) display time of solutions (s)	$\mathcal{A}$	10.40	14.60	4.04	3.20	3.33	.005	**
	$\mathcal{B}$	4.58	4.63	4.40	4.50	0.30	.613	

[a] $\mathcal{A}$: SPARQL, $\mathcal{B}$: SemwidgQL
*$p < .05$, **$p < .01$, ***$p < .001$

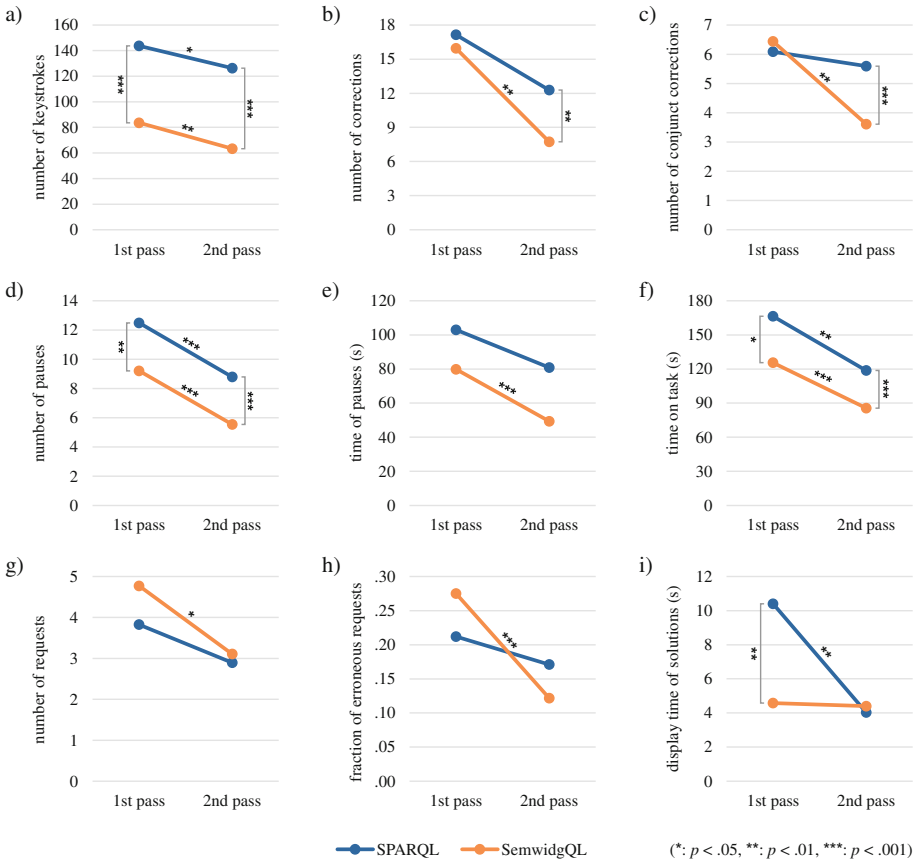

Fig. 3. Differences between SPARQL and SemwidgQL per pass, and differences between first and second pass per language.

measure, which is based on the number of nodes of a query in SPARQL Syntax Expressions (SSE) notation[3]. Later on, we show that the empirical data are better represented by the alternative SSE based complexity measure.

To calculate the SSE based complexity measure, we summed up the number of nodes of the SSE syntax tree, but combined all nodes which were required for matching the language in a filter expression into one. Since the participants were taught this filter as a fixed expression in both languages, we assumed that writing this expression requires no additional mental effort than a normal filter expression. Also, we did not count the first projection node (i.e. SELECT), which occurs in all SELECT queries. Table 5 shows the complexity values of the SPARQL sample solutions of each task in comparison, calculated according to Halstead's D as well as the SSE based complexity c. D and c values of tasks without filter

[3] https://jena.apache.org/documentation/notes/sse.html.

Table 5. Comparison of complexity of SPARQL sample solutions.

	Task								
	4	5	6	7	8	9	10	11	12
Halstead D	2.67	4.00	3.50	8.25	3.60	7.71	3.75	8.40	9.10
SSE based complexity c	2	3	3	4	3	6	3	5	8

expressions (4, 5, 6, 8, 10) are very similar, while D values of tasks with filter expressions (7, 9, 11, 12) are noticeably higher than c values. We argue that this method is much closer aligned to the mental processes a user has to perform when solving a task than Halstead's method. We calculated the regression lines for all response variables with each D and c as predictors and SPARQL as query language. Based on the yielded coefficient of determination R^2, we calculated a Wilcoxon Signed-Rank Test that supports our statement and indicates that the median for c, $Mdn = .86$, was significantly better than the median for D, $Mdn = .56$ ($z = -2.35$, $p = .016$).

The results of the linear regression analyses with c as predictor variable for all response variables with SPARQL and SemwidgQL are presented in Table 6 and Fig. 4. Since we did not want to compare the theoretical complexity of SPARQL and SemwidgQL but their practical performance at tasks with different complexities, we chose the complexity value of the SPARQL sample solution query

Table 6. Linear regression analyses with SSE based complexity as predictor.

	Lang[a]	$F(1,7)$	p	R^2	f	$f_i(c)$[b]
(a) number of keystrokes	$\mathcal{A}$	92.43	<.001	.93	3.63	$23.90c + 41.98$
	$\mathcal{B}$	64.04	<.001	.90	3.02	$21.32c - 9.57$
(b) number of corrections	$\mathcal{A}$	36.93	.001	.84	2.30	$4.00c - 1.30$
	$\mathcal{B}$	24.22	.002	.78	1.86	$4.33c - 4.95$
(c) number of conjunct corrections	$\mathcal{A}$	42.98	<.001	.86	2.48	$1.67c - 0.77$
	$\mathcal{B}$	25.06	.002	.78	1.89	$1.82c - 2.13$
(d) number of pauses	$\mathcal{A}$	156.63	<.001	.96	4.73	$3.08c - 0.72$
	$\mathcal{B}$	81.27	<.001	.92	3.41	$3.24c - 5.02$
(e) time of pauses (s)	$\mathcal{A}$	34.19	.001	.83	2.21	$39.51c - 50.15$
	$\mathcal{B}$	60.02	<.001	.90	2.93	$36.70c - 71.90$
(f) time on task (s)	$\mathcal{A}$	92.33	<.001	.93	3.63	$46.19c - 31.83$
	$\mathcal{B}$	57.90	<.001	.89	2.88	$40.24c - 47.17$
(g) number of requests	$\mathcal{A}$	18.27	.004	.72	1.62	$2.54c - 5.95$
	$\mathcal{B}$	59.15	<.001	.89	2.91	$2.18c - 3.41$
(h) fraction of erroneous requests	$\mathcal{A}$	2.05	.195	.23	0.54	$0.05c + 0.07$
	$\mathcal{B}$	0.52	.494	.07	0.27	$0.02c + 0.23$
(i) display time of solutions (s)	$\mathcal{A}$	19.27	.003	.73	1.66	$2.27c - 1.64$
	$\mathcal{B}$	7.41	.030	.51	1.03	$1.33c - 0.53$

[a] $\mathcal{A}$: SPARQL, $\mathcal{B}$: SemwidgQL
[b] Linear regression equation, i: measured response, c: complexity

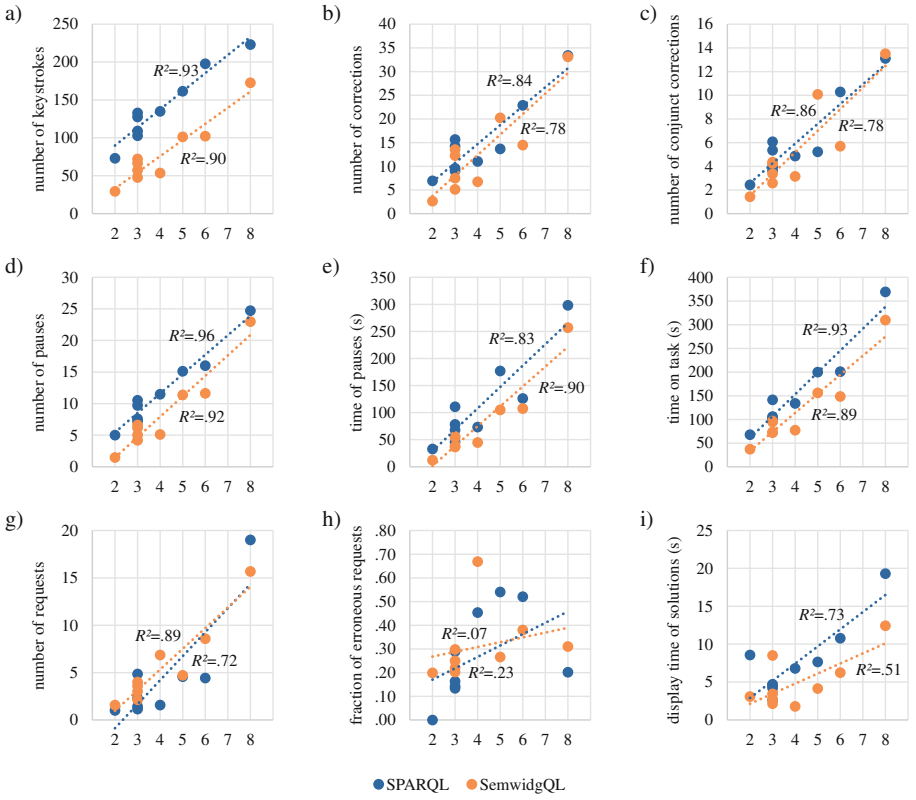

Fig. 4. Linear regression analyses with task complexity as predictor.

as complexity value for the corresponding tasks. Results indicate that there is a significant association between complexity c and all response variables except for *fraction of erroneous requests (h)*. The corresponding regression equations have very high R^2 values (one third of them $\geq .90$) and the effect sizes f are also high, according to Cohen [4].

The regression lines for the values of SemwidgQL in the examined range of c are in all cases, except *number of requests (g)* and *fraction of erroneous requests (h)*, below the regression lines for the values of SPARQL. In four of these seven cases, the slopes of the SemwidgQL lines are less steep than the slopes of the SPARQL lines, suggesting that SemwidgQL will perform better then SPARQL at more complex tasks. In the remaining three cases, the lines intersect at complexity values above the investigated range.

Subjective Evaluation by the Participants: In the following subsection, we will present the results of the questionnaire, the participants completed after the query tasks. We calculated multiple Wilcoxon Signed-Rank Tests to compare the participants' subjective ratings for SPARQL and SemwidgQL. *Writing effort* was

rated significantly better regarding SemwidgQL, $Mdn = 4$, compared to SPARQL, $Mdn = 2$ ($z = -3.03$, $p = .001$). *Sophistication* was also rated significantly better regarding SemwidgQL, $Mdn = 4$, compared to SPARQL, $Mdn = 2$ ($z = -3.23$, $p < .001$). There were no significant differences regarding the subjective ratings of *learnability*, *intuitiveness*, *logical structure*, and *comprehensibility* (see Fig. 5). When asked for advantages of SemwidgQL over SPARQL, the participants named nine unique characteristics with 30 occurrences in total. Particularly the shortness of queries and the similarity to object orientated programming languages were frequently mentioned. The participants only named three unique advantages of SPARQL over SemwidgQL with 5 occurrences in total (see Fig. 6). In 79% of the

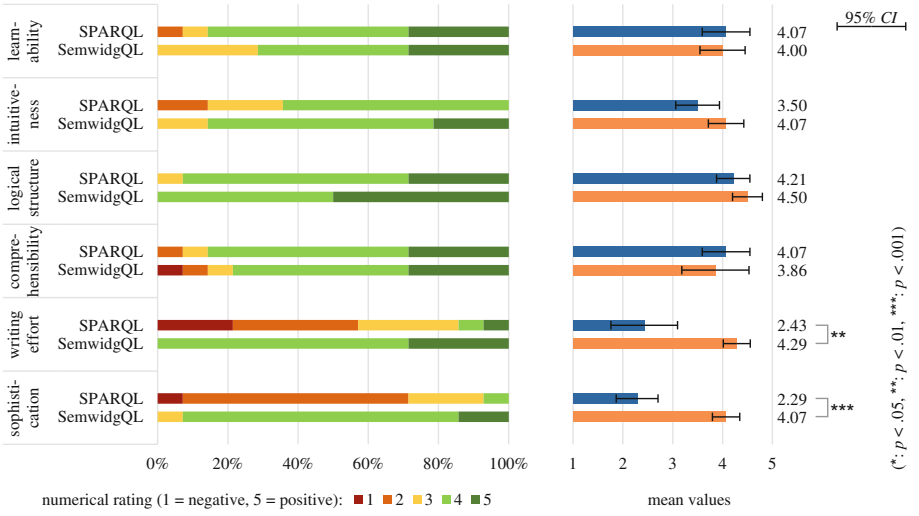

Fig. 5. Subjective evaluation of SPARQL and SemwidgQL.

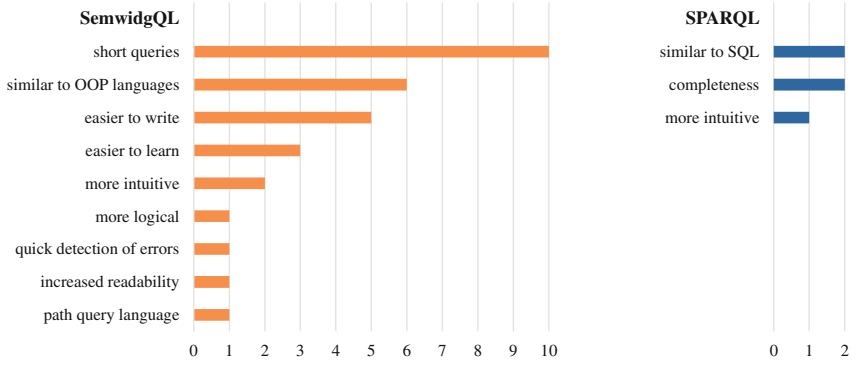

Fig. 6. Cumulative numbers of stated advantages of SPARQL and SemwidgQL.

answers SemwidgQL was named as the preferred query language. Accordingly, SPARQL was only preferred in 21% of the answers.

4.3 Discussion

The user study showed that the participants performed significantly better with SemwidgQL regarding most of the evaluated dependent variables compared to SPARQL. Especially the time on task, the number of corrections, and the number and time of pauses that the participants took to think about the correct solution of the task indicate that SemwidgQL is easier to use than SPARQL.

After the introductory session, the participants achieved better results with SemwidgQL than with SPARQL regarding most of the evaluated dependent variables. They improved significantly in the second pass in all but one area with SemwidgQL. Most of the improvements with SPARQL were not significant. The participants had already performed better in the first pass with SemwidgQL, and had improved even more in the second pass compared to SPARQL. The results suggest that SemwidgQL is easier to learn.

The reason for the better results with SemwidgQL was not that some already simple tasks were made even easier. The linear regression analyses indicate that the good results with SemwidgQL were achieved at all evaluated complexity levels. Some regression lines predict that even more complex tasks than that we have evaluated can be solved better with SemwidgQL. However, it should also be noted that some regression lines indicate that users will perform worse with SemwidgQL at tasks with higher complexity levels than evaluated. Since continuously written SemwidgQL queries can become very unwieldy at a certain length, this is to be expected.

The good results of the objective measures are supported by the participants' subjective evaluation, the number of mentioned advantages and, of course, the explicit personal preference for SemwidgQL of 79%.

5 Evaluation of SemwidgQL's Expressiveness

To investigate how well SemwidgQL covers the range of SPARQL queries used in practice we analyzed to what extent our language is able to express the queries that occur in the Linked SPARQL Queries Dataset (LSQ), collected by Saleem et al. [13]. We extracted 636,876 unique SELECT queries with 1,526,804 executions and then transformed them into a parameterized form. We mapped all IRIs, variables, literals and language tags of each query to a generic format (e.g. `SELECT ?v2 WHERE {<i1> ?v1 ?v2}`), and replaced all wildcards in SELECT statements with the corresponding list of variables from the WHERE statement and harmonized language filter expressions. Completely identically parameterized queries were merged automatically. We were able to manually merge further pattern that were not syntactically but semantically identical (e.g. queries with the same triple patterns in their WHERE clauses, but in different order, or queries with and without the DISTINCT keyword, where the DISTINCT keyword is not

able to reduce the result set). Finally, we obtained 1619 unique query patterns where the first 120 patterns of the most frequently executed queries represent 99% of the SELECT queries executed overall in the LSQ dataset.

Based on these 120 query patterns, we evaluated how well SemwidgQL covers the range of SPARQL queries used in practice. From these patterns 66, representing 91% of the overall executed queries, can be directly expressed in SemwidgQL without any limitations. In contrast, 15 of these patterns, representing only 2% of the overall executed queries, can not be expressed. These patterns contain GROUP BY expressions or function calls, such as `bound` or `isLiteral`, which are not implemented in SemwidgQL. The remaining 39 patterns, representing 6% of the queries executed overall, make use of UNION graph patterns. SemwidgQL does not provide an equivalent for these constructs. Nevertheless, some of these patterns can be expressed without UNION but with FILTER expressions. Additionally, SemwidgQL allows the declaration of multiple queries in a single statement. These queries are translated into separate SPARQL queries. Combining their results is up to the processing program.

We calculated the SSE based complexity measure for the 120 most frequently used query patterns. Most of the requests made (89%) have a c value below or equal to 8 and thus lay in the evaluated range of our user study. One third of them have a c value of 2 or 3. Few query patterns have c values above 20 (up to 58). However, these patterns only represent less than 3% of the requests made.

6 Conclusion

We have presented SemwidgQL, a path query language for RDF data that transcompiles to SPARQL. Our empirical user study indicates that SemwidgQL is easier to learn, more efficient, and preferred by the learners compared to SPARQL. An additional evaluation of the LSQ dataset indicates that SemwidgQL, despite its limited expressiveness, is capable of querying most of the data that is currently queried with SPARQL. Also, the queries we used in the user study have a comparable complexity to queries that are used in practice. SemwidgQL is not intended as a replacement for SPARQL but rather as a more light-weight language that lowers the entry barriers to the Semantic Web and Linked Data area. Results indicate that SemwidgQL is suitable for this purpose.

References

1. Barbieri, D.F., Braga, D., Ceri, S., Grossniklaus, M.: An execution environment for C-SPARQL queries. In: Proceedings of the 13th International Conference on Extending Database Technology, EDBT 2010, pp. 441–452. ACM, New York (2010)
2. Berners-Lee, T., Chen, Y., Chilton, L., Connolly, D., Dhanaraj, R., Hollenbach, J., Lerer, A., Sheets, D.: Tabulator: exploring and analyzing linked data on the semantic web. In: Proceedings of the 3rd International Semantic Web User Interaction Workshop (2006)
3. Casterella, G.I., Vijayasarathy, L.: An experimental investigation of complexity in database query formulation tasks. J. Inf. Syst. Educ. **24**(3), 211 (2013)

4. Cohen, J.: A power primer. Psychol. Bull. **112**(1), 155 (1992)
5. Corcho, O., Calbimonte, J.P., Jeung, H., Aberer, K.: Enabling query technologies for the semantic sensor web. Int. J. Semant. Web Inf. Syst. **8**(1), 43–63 (2012)
6. Halstead, M.H.: Elements of Software Science, vol. 7. Elsevier, New York (1977)
7. Harth, A.: VisiNav: a system for visual search and navigation on web data. Web Semant. Sci. Serv. Agents World Wide Web **8**(4), 348–354 (2010)
8. Heim, P., Hellmann, S., Lehmann, J., Lohmann, S., Stegemann, T.: RelFinder: revealing relationships in RDF knowledge bases. In: Chua, T.-S., Kompatsiaris, Y., Mérialdo, B., Haas, W., Thallinger, G., Bailer, W. (eds.) SAMT 2009. LNCS, vol. 5887, pp. 182–187. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10543-2_21
9. Lassila, M., Junkkari, M., Kekäläinen, J.: Comparison of two XML query languages from the perspective of learners. J. Inf. Sci. **41**(5), 584–595 (2015)
10. Leinberger, M., Scheglmann, S., Lämmel, R., Staab, S., Thimm, M., Viegas, E.: Semantic web application development with LITEQ. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8797, pp. 212–227. Springer, Cham (2014). doi:10.1007/978-3-319-11915-1_14
11. Pietriga, E., Bizer, C., Karger, D., Lee, R.: Fresnel: a browser-independent presentation vocabulary for RDF. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 158–171. Springer, Heidelberg (2006). doi:10.1007/11926078_12
12. Reisner, P., Boyce, R.F., Chamberlin, D.D.: Human factors evaluation of two data base query languages: square and sequel. In: Proceedings of the National Computer Conference and Exposition, AFIPS 1975, pp. 447–452. ACM, New York, 19–22 May 1975
13. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.-C.N.: LSQ: the linked SPARQL queries dataset. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 261–269. Springer, Cham (2015). doi:10.1007/978-3-319-25010-6_15
14. Schaffert, S., Bauer, C., Kurz, T., Dorschel, F., Glachs, D., Fernandez, M.: The linked media framework: Integrating and interlinking enterprise media content and data. In: Proceedings of the 8th International Conference on Semantic Systems, pp. 25–32. I-SEMANTICS 2012. ACM, New York (2012)
15. Schmachtenberg, M., Bizer, C., Paulheim, H.: Adoption of the linked data best practices in different topical domains. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 245–260. Springer, Cham (2014). doi:10.1007/978-3-319-11964-9_16
16. Stegemann, T., Ziegler, J.: SemwidgJS: a semantic widget library for the rapid development of user interfaces for linked open data. In: Plödereder, E., Grunske, L., Schneider, E., Ull, D. (eds.) 44. Jahrestagung der Gesellschaft für Informatik GI, Informatik 2014. LNI, vol. 232, pp. 479–490 (2014)