UniQue: An Approach for Unified and Efficient Querying of Heterogeneous Web Data Sources

Markku Laine, Jari Kleimola and Petri Vuorimaa
Department of Computer Science, Aalto University, Espoo, Finland

Keywords: Mashup Applications, Web Querying, Web Standards, XML Technologies.

Abstract: Governments, organizations, and people are publishing open data on the Web more than ever before. To consume the data, however, requires substantial effort from web mashup developers, as they have to familiarize themselves with a diversity of data formats and query techniques specific to each data source. While several solutions have been proposed to improve web querying, none of them covers aforementioned aspects in a developer friendly and efficient manner. Therefore, we devised a unified querying (UniQue) approach and a proxy-based implementation that provides a uniform and declarative interface for querying heterogeneous data sources across the Web. Besides hiding the differences between the underlying data formats and query techniques, UniQue heavily embraces open W3C standards to minimize the learning effort required by developers. Pursuing this further, we propose Unified Query Language (UQL) that combines the expressiveness of CSS Selectors and XPath into a single and flexible selector language. We show that the adoption of UniQue and UQL can effectively streamline web querying, leverage developers' existing knowledge, and reduce generated network traffic compared to the current state-of-the-art approach.

1 INTRODUCTION

Traditional database experts rely on well-defined data formats and query language APIs. The query API, such as SQL, provides precisely focused access to the data. Although the persistent data and its structure vary between development projects, the data format and the query API remain the same. Database experts are thus able to leverage their previous experience, tools, and reusable code snippets from past development efforts.

Web mashup developers (hereinafter referred to as "developers") are less fortunate. First, web data is served in a variety of data formats, and it is not uncommon to mix and match XML, JSON, or even HTML and CSV in a single mashup application. Second, to access the data, developers need to conform to several proprietary APIs. This requires (re-)learning, since even if data providers offer RESTful APIs, the query string parameters are likely to differ. The APIs are often also inefficient and return substantial amount of redundant data, which needs to be filtered on the client side with yet another API.

While semantic web technologies (e.g., RDF and SPARQL) have been proposed (Harth et al., 2011) as a solution to uniform data access, they are not broadly adopted due to their inherent complexity and unfamiliarity among developers. Indeed, the majority of web data providers are still relying on more mainstream data formats and developers on less complex selector languages.

Therefore, we propose a unified web querying approach that a) builds on these prevailing practices and b) takes full advantage of developers' existing experience with standard web technologies and associated tools. This approach, which we call *UniQue*, provides a uniform and declarative query interface that allows developers to perform precise selection queries against heterogeneous data on the Web. We focus on efficient querying of text-based data originating from web data sources and services (e.g., static files and Web APIs, respectively). Querying of big data sets and databases is beyond the scope of this work.

The main contributions of this paper are:

- We propose an approach (UniQue) for unified and efficient querying of heterogeneous web data sources.
- We propose improvements to the guidelines for mapping JSON and CSV data into a single form (XML); a special focus is put on

friendliness and round-trippability.

- We propose a unified query language (UQL), which extends CSS Selectors with XPath expressions for improved expressiveness.
- We report on the results of an experimental study, in which we compared the UniQue approach with the current state-of-the-art approach.

The rest of this paper is organized as follows. Section 2 presents a motivating example of querying heterogeneous web data sources and the challenges related to it. Section 3 defines the requirements for a unified and efficient web querying approach. Section 4 introduces the design of the proposed approach, while Section 5 describes our proxy-based implementation. Section 6 evaluates the approach and presents the results with discussion. Section 7 compares our approach to related work and Section 8 draws conclusions.

2 MOTIVATING EXAMPLE AND CHALLENGES

As a motivating example, consider *Alice*, who is a developer and an avid climber. Alice has decided to build a web mashup application that provides a central place for climbers alike to stay updated on all the latest from elite rock climbers.

Figure 1 shows a basic design for Alice's mashup application. According to the design, the application needs to connect to the total of eight web data sources in order to retrieve all the required data. Moreover, the data sources are completely different from each other, as shown in Table 1. First, Alice needs to scrape data from *hardclimbs.info* to generate a list of world's top 10 climbers, and later to retrieve their hardest ascents. She also needs to read climbers' social media account IDs and competition results from static files. This requires writing custom data wrappers or data transformations. Finally, to retrieve climbers' basic information, tweets, and photos from social media services, Alice needs get familiar with the documentation and query parameters of each Web API endpoint.

2.1 Challenges

We identified three major challenges that developers (i.e., Alice) face in building web mashup applications.

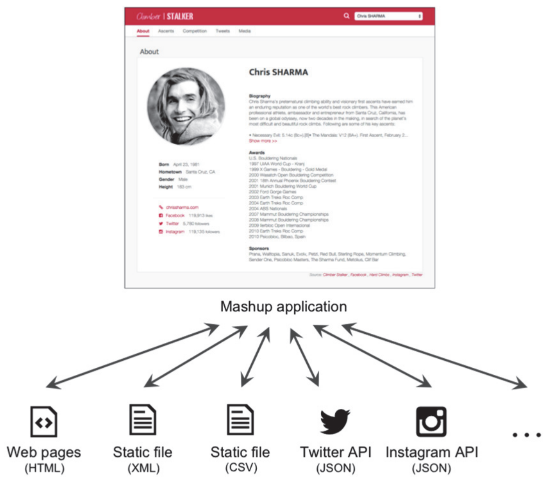


Figure 1: Web mashup application accessing multiple web data sources and services.

Table 1: Information about the data sources used by the web mashup application.

Data Source	Provider	Description	Type	Output Format
I	Hard Climbs	Top 10 climbers	Web page	HTML
II	Climber Stalker	Climber account IDs	Static file	XML
III	Facebook	About	Web API	JSON
IV	Instagram	About	Web API	JSON
V	Hard Climbs	Ascents	Web page	HTML
VI	IFSC	Competition	Static file	CSV
VII	Twitter	Tweets	Web API	JSON
VIII	Instagram	Media	Web API	JSON

2.1.1 Challenge 1: Heterogeneous Data Formats

Web data exists in many different formats. As a result, Alice needs to learn and understand the differences and similarities between multiple data models, and deal with the impedance mismatch problem.

2.1.2 Challenge 2: Heterogeneous Query Techniques

The heterogeneity of web data formats leads to a situation where Alice needs to master a diverse range of query techniques, including web scraping, writing custom data wrappers, and consuming Web APIs. Each query technique in turn involves specific technologies and specifications that Alice needs to get familiar with.

2.1.3 Challenge 3: Inefficient Use of Network Resources

Many web data sources (e.g., static files, web pages, and web feeds) lack of an API. Therefore, Alice needs to pull the full response data to the client, even if all that she needs is a fraction of that data. Web

APIs partially suffer from the same problem, as their filtering capabilities are usually rather limited.

In summary, these challenges not only decrease Alice's productivity as a developer, but also increase the application complexity for data access and querying.

3 REQUIREMENTS

Based on our motivating example and the literature review (cf. Section 7), we derived a list of requirements that fall within the scope of this paper.

3.1 Technology Requirements

R1: **Ease of Authoring.** Technologies must be familiar to a broad range of developers. Generally, declarative languages are considered easier to reason about than imperative ones (Van Roy and Haridi, 2004), and thus should be preferred.

R2: **Web Integration.** Technologies must be open and standardized to enable interoperability with existing tools, broad adoption, and future uses on the Web.

3.2 Data Format Requirements

R3: **Uniform Data Representation.** Heterogeneous web data must be converted into a single, human and machine-readable format in order to minimize the knowledge required by developers.

R4: **Friendly and Round-Trippable Mappings.** Mapping original data to a single format must produce data that is both easy to consume and query, i.e., friendly. Additionally, it must be possible to map the data back to its original format, i.e., round-trippable, for data updates. Focus should be put on friendliness, while preserving round-trippability.

3.3 Data Access and Query Requirements

R5: **Uniform Data Access.** Heterogeneous web data sources must be accessible and queryable through a single interface (e.g., Web API) in order to minimize the learning effort required by developers. Moreover, the interface must provide a means of executing queries also against web data sources without available APIs.

R6: **Flexible Query Language.** The query language must be simple yet expressive enough for retrieving data of interest (less data to be transferred). Moreover, it must be possible to extend its expressiveness while keeping the learning curve to a minimum (MacLean et al., 1990).

3.4 System Requirements

R7: **Client Independence.** The system must support various clients (e.g., web browsers and HTTP client software/libraries), without requiring any additional software installation.

R8: **Server-Side Processing.** The system must perform all processing on the server side in order to elicit reductions in response data.

R9: **Data Source Server Independence.** The system must support existing web data source servers, without requiring any modifications to them.

4 THE UNIQUE APPROACH

In this section, we present our unified querying approach to meet the requirements presented in Section 3. We set out to give an overview of the proposed approach and continue with a detailed description of its essential parts.

4.1 The Approach in a Nutshell

We propose the *UniQue* approach to help developers in accessing and querying JSON / CSV / HTML / XML data originating from heterogeneous web data sources, such as static files and Web APIs. The design of the proposed approach fulfills the requirements R1-R6, while the system implementation focuses on the requirements R7-R9.

The basic idea is to expose a uniform query interface (cf. R5) through which the developers can explore target data and perform selection queries against it. Consequently, only data of interest is retrieved. The interface is based on markup languages (XML) and element selection queries (CSS Selectors); the core concepts and declarative technologies that web developers learn from day one (cf. R1). Besides being open W3C standards (cf. R2), both of the technologies are expressive and flexible enough to serve as a basis for the rest of the design requirements of our approach.

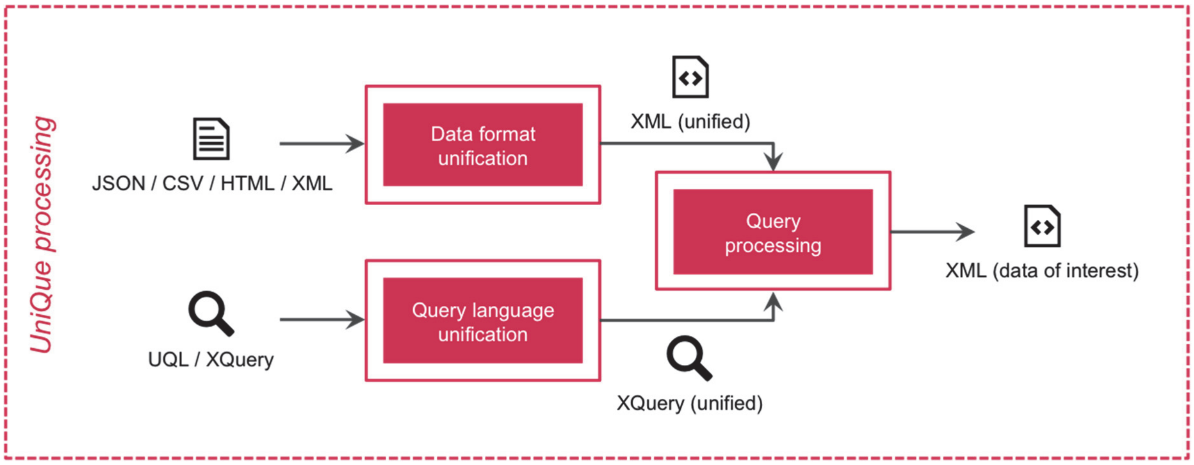


Figure 2: Overview of the UniQue processing.

The overview of the UniQue processing shown in Figure 2 crystallizes the concepts and technologies behind the proposed approach. On receiving the data and the selection query, the data goes through a unification process. In the *Data format unification* process (details in Section 4.2), JSON / CSV / HTML / XML data is converted into friendly and round-trippable XML by following a set of mapping rules. In *Query language unification* (details in Section 4.3), a similar conversion process is applied to a selection query written in UQL / XQuery to translate it into XQuery. Finally, in the *Query processing* phase, the resulting XQuery (unified) is evaluated and executed against the resulting XML (unified) to produce output XML, which includes only data of interest.

4.2 Data Format Unification

In the following subsections, we provide guidelines for mapping varying data formats into a single form. We chose XML as the unified data format because it is more verbose than JSON or CSV. Data mappings minimize developers' effort, since they need to master only a single data format and related tools/libraries (cf. R3). While studying the mappings, a special focus was put on two major aspects, *friendliness* (primary) and *round-trippability* (secondary), identified by (Boyer et al., 2011) (cf. R4).

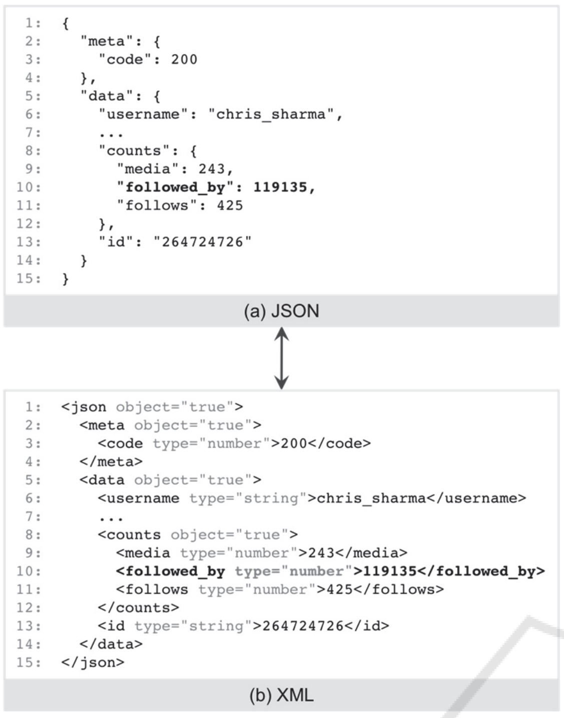
4.2.1 JSON to XML

For mapping JavaScript Object Notation (JSON) (Bray, 2014) to XML, we used the mapping rules described in the forthcoming W3C XForms 2.0 specification[1] (Boyer et al., 2016). Briefly, each JSON name/value pair becomes an XML element, whose name is the JSON name and whose value is the JSON value. In the case of a JSON array, a separate XML element following the afore-mentioned rules is created for each value of the array. Additionally, XML attributes are added to the element to store the type of the JSON value along with other metadata required for round-trippability. In case round-trippability is not required, i.e., the data is meant for consumption only, the attributes can be omitted.

We contributed to the design of these mapping rules in earlier revisions. Specifically, we pointed out deficiencies regarding round-trippability and suggested improvements.

Figure 3 gives an example of the JSON to XML mapping rules. In the example, (a) a JSON response and (b) its XML equivalent are shown when calling the Instagram API method "`users/{user-id}`" to get basic information about a user, in this case, a climber named Chris Sharma. As can be seen, the resulting XML has meaningful element names, making the data easy to query (i.e., friendly). Further, the type and order information of the original JSON are preserved in attributes for round-trippability.

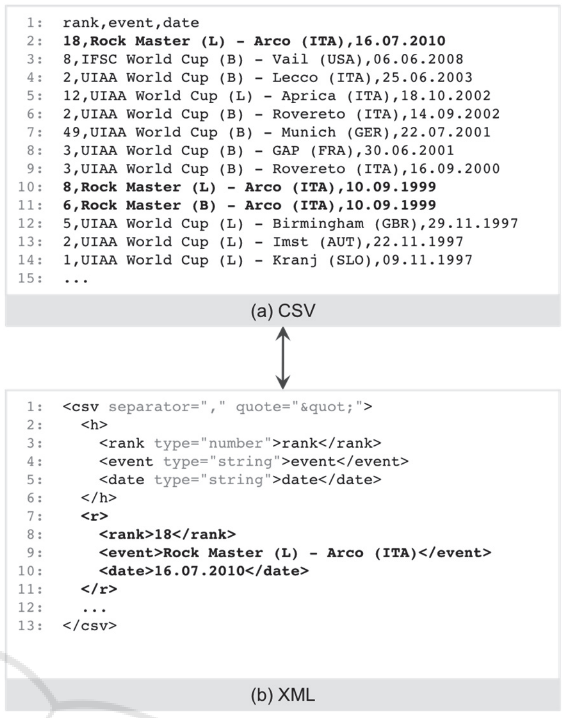
[1] Revision as of March 18, 2015



```
 1: {
 2:   "meta": {
 3:     "code": 200
 4:   },
 5:   "data": {
 6:     "username": "chris_sharma",
 7:     ...
 8:     "counts": {
 9:       "media": 243,
10:       "followed_by": 119135,
11:       "follows": 425
12:     },
13:     "id": "264724726"
14:   }
15: }
```

```
 1: <json object="true">
 2:   <meta object="true">
 3:     <code type="number">200</code>
 4:   </meta>
 5:   <data object="true">
 6:     <username type="string">chris_sharma</username>
 7:     ...
 8:     <counts object="true">
 9:       <media type="number">243</media>
10:       <followed_by type="number">119135</followed_by>
11:       <follows type="number">425</follows>
12:     </counts>
13:     <id type="string">264724726</id>
14:   </data>
15: </json>
```

Figure 3: Example of mapping data between (a) JSON and (b) XML. The data of interest in line 10 is bolded (referred later in the paper), whereas attributes required for round-trippability are de-emphasized in gray.



```
 1: rank,event,date
 2: 18,Rock Master (L) - Arco (ITA),16.07.2010
 3: 8,IFSC World Cup (B) - Vail (USA),06.06.2008
 4: 2,UIAA World Cup (B) - Lecco (ITA),25.06.2003
 5: 12,UIAA World Cup (L) - Aprica (ITA),18.10.2002
 6: 2,UIAA World Cup (B) - Rovereto (ITA),14.09.2002
 7: 49,UIAA World Cup (B) - Munich (GER),22.07.2001
 8: 3,UIAA World Cup (B) - GAP (FRA),30.06.2001
 9: 3,UIAA World Cup (B) - Rovereto (ITA),16.09.2000
10: 8,Rock Master (L) - Arco (ITA),10.09.1999
11: 6,Rock Master (B) - Arco (ITA),10.09.1999
12: 5,UIAA World Cup (L) - Birmingham (GBR),29.11.1997
13: 2,UIAA World Cup (L) - Imst (AUT),22.11.1997
14: 1,UIAA World Cup (L) - Kranj (SLO),09.11.1997
15: ...
```

```
 1: <csv separator="," quote="&quot;">
 2:   <h>
 3:     <rank type="number">rank</rank>
 4:     <event type="string">event</event>
 5:     <date type="string">date</date>
 6:   </h>
 7:   <r>
 8:     <rank>18</rank>
 9:     <event>Rock Master (L) - Arco (ITA)</event>
10:     <date>16.07.2010</date>
11:   </r>
12:   ...
13: </csv>
```

Figure 4: Example of mapping data between (a) CSV and (b) XML. The data of interest in lines 7–11 is bolded (referred later in the paper), whereas attributes required for round-trippability are de-emphasized in gray.

4.2.2 CSV to XML

The forthcoming XForms 2.0 specification (Boyer et al., 2016) also describes mapping rules for converting Comma-Separated Values (CSV) (Shafranovich, 2005) data into friendly and round-trippable XML. Briefly, each label in the header row (if present) becomes an XML element whose name and value are same as the label. Illegal characters in the XML element name are replaced (or prepended in case of the first character) with the underscore character "_". Additionally, all these XML header elements are enclosed by an `<h>` element. Then, each field value in the (subsequent) record rows is mapped to an XML element whose name is the corresponding label (or `<v>`, in case headers are absent) and whose value is the corresponding field value. The XML record elements of each row are enclosed by an `<r>` element.

We extended the mapping rules to better support both round-trippability and differences among implementations generating CSV data. Specifically, we added two attributes to the root element of the resulting XML: *separator* and *quote*. Their values define the character to separate fields and the character to quote fields on header and record rows, respectively. In addition, we propose to add the *type* attribute to converted XML header elements to store the type of values in each CSV column. While adding the type information may not improve round-trippability (all field values in CSV are treated as strings), it increases the semantic level of data.

Figure 4 shows an example, in which (a) CSV data containing information about Chris Sharma's competition results is converted into (b) XML by following the above-described mapping rules.

4.2.3 Other Mappings

The rules for converting a HyperText Markup Language (HTML) (Hickson et al., 2014) document into a well-formed XML document (namely, XHTML) are already well established and follow a small set of guidelines defined in (Pemberton et al., 2002). These guidelines require that, for instance, all elements must have a closing tag and attribute values must be quoted. The resulting XML cannot be converted back into its original HTML form. However, all relevant information is preserved during the conversion process, as it only involves tidying up the HTML markup.

Since Extensible Markup Language (XML)

(Bray et al., 2008) data is already represented in well-formed XML, there is no need to apply any mappings.

4.3 Query Language Unification

The following subsections present UQL and XQuery. The former is a flexible selector language, whereas the latter supports more demanding query scenarios. We describe their commonalities and the unification process. We also show how developers can make a smooth transition from one language to another without introducing major learning barriers (cf. R6).

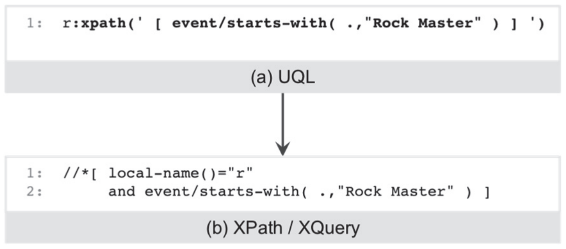
4.3.1 Unified Query Language

In this paper, we propose *Unified Query Language (UQL)* to efficiently retrieve (and transfer) only data of interest from target data sources. UQL is based on widely adopted W3C Selectors (Çelik et al., 2011) that developers use, for instance, to bind style properties to elements in CSS and match a set of elements in a document with W3C Selectors API (e.g., `querySelectorAll()`) or jQuery[2]. To give an example of Selectors' functionality, consider a scenario, in which the number of followers needs to be shown next to the Instagram icon. With the selector "`followed_by`", we can easily pull the desired data bolded in line 10 of Figure 3b. Obviously, Selectors allow much more sophisticated queries, as they can be combined and joined in many ways to achieve great specificity. However, Selectors have certain limitations, such as lack of content matching and cumbersome range selectors.

UQL addresses these limitations and improves the expressiveness of Selectors by extending it with a new functional pseudo-class `:xpath()`. With our extension, developers can make a natural and easy switch to similar but more expressive XPath (Robie et al., 2014a)—another widely adopted W3C selector language—whenever Selectors' expressiveness is not powerful enough. In other words, the extension exposes the full potential of XPath, including its functions, to developers. Consequently, UQL can cover a wide variety of use cases and requires only minimal learning effort from developers already familiar with Selectors.

We demonstrate the usefulness of UQL in a simple example, in which we want to select only those Chris Sharma's competition results from Figure 4b that took place at "Rock Master". By

[2] jQuery, http://jquery.com/



```
r:xpath(' [ event/starts-with( .,"Rock Master" ) ] ')
```

```
//*[ local-name()="r"
    and event/starts-with( .,"Rock Master" ) ]
```

Figure 5: Example of translating a query from (a) UQL to (b) XPath / XQuery. Our `:xpath()` extension in UQL is bolded.

using our extension with the UQL query "`r:xpath('[ event/starts-with( ., "Rock Master" ) ]')`" we discover that he has participated the competition three times (2010, 1999, 1999).

In our approach, UQL queries are translated to XPath equivalents for query processing. Hence, the semantics of UQL follow the semantics of XPath. Mapping Selectors parts to XPath is straightforward as both languages use path-based expressions and are syntactically very similar, as shown in Figure 5.

4.3.2 XQuery

For completeness and more demanding scenarios, we also support XQuery (Robie et al., 2014b). XQuery is a Turing-complete query language designed by the W3C for extracting and manipulating data from any data source that can be viewed as XML. XQuery extends XPath, so developers can transfer their knowledge gained from UQL to XQuery when writing queries. XQuery also supports the missing features of UQL, such as data grouping and sorting. Its increased expressiveness, however, comes at the expense of added complexity.

5 UNIQUE SYSTEM IMPLEMENTATION

In the following, we discuss the concrete implementation of the UniQue approach. Specifically, this section details the system architecture and its operation as well as the individual components realizing it.

5.1 System Architecture

Figure 6 illustrates the architecture of the UniQue system implementation. In our architecture, the UniQue system operates as a *proxy server* between the client (e.g., a web browser) and the data source servers (e.g., Web APIs, web pages, JSON / CSV / XML files, and RSS / Atom feeds). The proxy was implemented in XQuery 3.0 on top of the 28.io[3] platform. The platform itself is based on Zorba[4], which supports XQuery / JSONiq and other XML technologies relevant to our approach. The communication with the proxy takes place by making HTTP calls to its API.

We chose a proxy-based implementation for three reasons. First, a wide variety of clients can take advantage from the services it exposes. Second, it can significantly reduce network traffic between the client and the proxy. Third, it does not require any modifications to existing data source servers. Thus, the system conforms to the requirements R7-R9 outlined in Section 3.4.

5.2 Operation and Components

Figure 6 depicts the components and processing steps involved when the client invokes the UniQue Web API. The process starts by forming the URL for an HTTP GET request, which consists of an endpoint URL and query string parameters, such as *data*, *format*, and *query*.

Below is an example URL,

```
http://unique.28.io/processor.xq?
data={data}&
format={format}&
query={query}
```

where `data` points to a single target data source (absolute URL or inline text), `format` indicates its output format (e.g., json), and `query` holds a query expression in UQL in order to retrieve data of interest from the target data source. The HTTP GET request, i.e., UniQue query, is then sent to the proxy (1), where the *UniQue processor* reads the query string parameters and makes an HTTP GET request to retrieve the entire original data from the given URL (2). After receiving the result of the original response (3), the processor invokes the *Data converters* and *Query converters* modules to perform data and query unification, respectively. Next, the unified query is evaluated and executed against the unified data, yielding result data in XML (4). In case the client supports data compression, the proxy compresses the produced data before returning the result to the client (5). As a result of the whole process, the response data now contains only data of interest in a compressed form, which may reduce network traffic significantly.

[3] 28msec, http://www.28.io/

[4] Zorba, http://www.zorba.io/

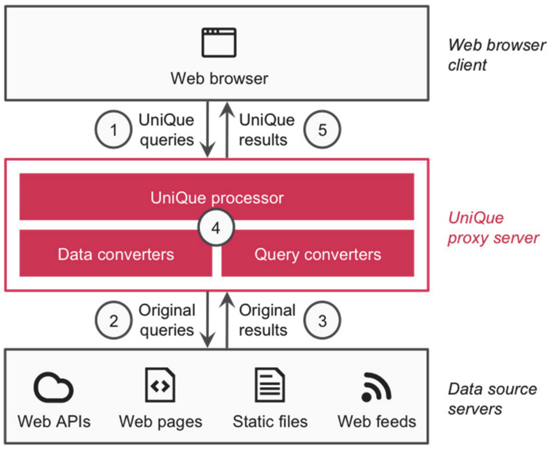


Figure 6: Architecture of the UniQue system implementation.

5.2.1 UniQue Processor

The UniQue processor is the main component that is responsible for reading inputs, processing queries, and returning results. It invokes the Data converters and Query converters modules.

5.2.2 Data Converters

The data converters module provides functions necessary to parse input data (a string) and convert it into XML. The module contains four public functions—one for each data format supported, i.e., JSON, CSV, HTML, and XML—that implement the mappings discussed in Section 4.2. All the four function implementations use a dedicated built-in Zorba function as a basis for their data conversion. In the case of JSON to XML and CSV to XML, the built-in Zorba functions by themselves were incapable of directly producing the desired XML structure. Therefore, an additional data processing phase is applied with those two data conversions to transform a generic, *unfriendly* XML result generated by the Zorba functions into the desired, *friendly* XML representation format. To the best of our knowledge, this module provides the first and improved implementation of the XForms 2.0 conversion rules for mapping data between JSON and XML as well as CSV and XML.

5.2.3 Query Converters

The query converters module provides a public function to parse input query (a string) and convert it into an XQuery expression, as defined in Section 4.3. First, the function parses the input query with an assumption that the syntax follows a case-sensitive Extended Backus-Naur Form (EBNF) grammar of UQL 1.0 / Selectors 3.0 (both have the same grammar). The parser's XQuery code was generated semi-automatically from the EBNF grammar using the REx[5] parser generator (version 5.30). After successfully finishing the parsing process, the parser returns an XML parse tree. Next, the returned XML is parsed, and as a result, the input query given in UQL 1.0 / Selectors 3.0 is translated into an XPath 1.0 expression. Finally, an XQuery 3.0 expression is constructed by prepending the XPath expression with version and given namespace declarations. In case the parser fails to parse the input query, the function assumes that the syntax is XPath 1.0 / XQuery 3.0 compliant and prepends namespace declarations if provided.

5.3 Availability

The UniQue system implementation is made available to the public under the MIT license. The latest source code release can be downloaded from the project website at `https://mediatech.aalto.fi/publications/webservices/unique/`.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate *UniQue* with respect to the challenges identified in Section 2.1. Our evaluation also includes a comparison against a similar proxy-based approach called *YQL* (Yahoo, 2016), which represents the current state-of-the-art. Next, we describe our experimental design and results, followed by a brief discussion.

6.1 Experimental Design

We developed two versions of Alice's web mashup application depicted in Figure 1; one using UniQue and one using YQL for accessing the data sources (eight in total) shown in Table 1. To compare the approaches in terms of used data formats and query techniques, and to measure generated network traffic, we extracted the related data source queries from both applications. Each data source was then queried separately by sending an HTTP GET request from Chrome 45 web browser running on Mac OS X 10.8.5 with 3.06 GHz Intel Core 2 Duo processor and 4 GB of RAM over a wireless network connection. The proxies and data sources were running on third-party servers. To ensure a fair comparison, we requested both proxies to return response data in a format as similar as possible, i.e., XML and without attributes required for round-trippability. For capturing the browser's network traffic with the proxies, we used the HTTP Archive (HAR)[6] format.

The data sets, queries, and their associated evaluation results are all available at the project website.

6.2 Results

6.2.1 Data Formats

As shown by the last column of Table 1, the original data sources used varying data formats to return data. Nevertheless, both proxies succeeded in converting all the data retrieved from the original data sources into well-formed XML. There were no major differences between the conversion results; the resulting data appeared natural and used elements only (as opposed to having relevant data placed within attributes), making it easy to work with. From the developer's perspective, we observed that consuming the data required no or very little understanding of data formats other than XML.

6.2.2 Query Techniques

In both approaches, the data from the original data sources was accessed through the proxy's own Web API (endpoint URL with appropriate parameters). To select exactly the data of interest, a query expression was passed as a query parameter in the URL. With UniQue, the expressiveness of UQL was found sufficient in 7 out of 8 queries. Specifically, the use of pure Selectors was enough in five queries, while the proposed `:xpath()` extension was needed in two queries to match against element values. The more expressive XQuery was leveraged with Data Source I to construct a highly specific XML output from the original data. YQL in turn used SQL-like query statements accompanied with dot-style syntax as a basis for filtering the data. In 5 out of 8 queries,

[5] REx Parser Generator, http://bottlecaps.de/rex/

[6] HTTP Archive 1.2, http://www.softwareishard.com/blog/har-12-spec/

the syntax was found sufficient. With the three remaining queries, there was a need for a supplementary selector language, i.e., Selectors or XPath, within an SQL statement. Additionally, XSLT was needed to query Data Source I.

In conclusion, the main difference between the two approaches was in the selection of query languages. UniQue leveraged web mashup developers' prior knowledge on W3C-standardized Selectors and provided a gentle slope to extend its expressiveness. YQL in turn relied on SQL-like syntax, which is more familiar to database experts, and other rather different query languages to increase its expressiveness.

6.2.3 Network Traffic

Figure 7 presents the results of our comparative performance study regarding generated network traffic. From the figure we clearly see that the generated network traffic was significantly smaller with UniQue; 2 115 bytes on average compared to 4 332 bytes of YQL (about 51% reduction ratio). The high reduction ratio mainly results from the fact that our proxy compresses HTTP response body data whenever the requesting client supports it[7], whereas the YQL proxy never does so. Moreover, even without the effects of additional gzip compression on our proxy, the generated network traffic would still have been about 3% smaller with UniQue compared to YQL.

6.3 Discussion

The results from our experimental evaluation suggest that UniQue is a promising approach for querying heterogeneous web data sources. The benefits of adopting our approach were particularly apparent in terms of effective use of developers' prior knowledge on W3C standards and reduced network traffic through extendable query capabilities and data compression. Additionally, the process of web querying was simplified. We believe that these results will be of particular interest to the XForms community.

Our evaluation case study also revealed potential areas of improvement for the proposed UniQue approach. For instance, UQL could natively cover most typical use cases for the `:xpath()` extension in the future, such as matching against element values. Many of the current shortcomings of

[7] All modern web browsers support HTTP compression.

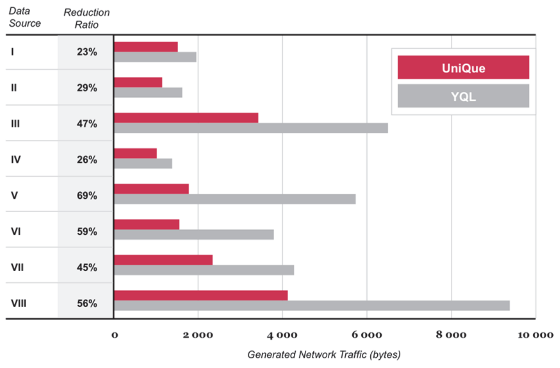


Figure 7: Comparison of generated network traffic per data source between UniQue and YQL.

Selectors (Level 3)—which UQL uses as a basis—will also be addressed in its future specifications (Etemad and Atkins, 2016; Kosek and Atkins, 2016).

We also noted that web mashup applications using our proposed approach could be complemented with a web performance solution, such as (Akamai, 2016; Google, 2016). These solutions provide automated web content and connection optimizations, such as image transcoding and HTTP multiplexing. As a result, the browser's overall network traffic can be reduced even further. Further, we believe that potential savings in data transfers might be of special interest for end users behind a slow network connection or on mobile, especially if they have limited bandwidth and/or a monthly data quota.

7 RELATED WORK

Over the years, numerous web query languages and techniques have been proposed in the literature; cf. Bailey et al. (2005) for a comprehensive survey. More recently, efforts have also been made to query newer web data formats, such as JSON. One of the most prominent query languages for it is JSONiq (Florescu and Fourny, 2013). JSONiq borrows several ideas from XQuery, such as a powerful FLWOR (For, Let, Where, Order by, and Return) construct as well as a declarative and functional style of programming. OXPath (Furche et al., 2013) in turn focuses on effectively scraping data from complex web applications. As its name suggests, the technique is based on an extended XPath language that allows declarative interaction with scripted HTML documents and the extraction of data from them. Giribet's (2005) proposal is another example of using XPath in querying web content. The technique combines XPath with URLs. Specifically, it uses XPath in the fragment identifier of an URL to explicitly specify the parts of XML data that are of interest. Similarly, Hausenblas et al., (2014) uses fragment identifiers with tailor-made methods to select specific rows, columns, or cells from CSV data. The main disadvantage of using fragment identifiers lies, however, in their inefficient data delivery. Namely, fragment identifiers are only interpreted by the client upon first receiving full response data. In contrast, UniQue performs data filtering on the proxy server and returns only data of interest to the client, and thus reduces network traffic. Additionally, it supports querying of all the above-mentioned data formats through a single, uniform, and declarative query interface.

According to Bischof et al. (2012), most of the existing approaches for querying heterogeneous data formats can be divided into two categories: data translation and language integration. In *data translation*, data is transformed from one representation format into another using predefined mapping rules. For example, 28.io (28msec, 2016) provides proprietary XQuery / JSONiq functions for converting data between different formats, including XML, JSON, HTML, and CSV. Boyer et al. (2011) go beyond straightforward data conversions of this nature and discuss different mapping approaches between XML and JSON from the aspects of friendliness and round-trippability. The same aspects are also considered important in the design of XForms 2.0 (Boyer et al., 2016) mappings used in UniQue. In *language integration*, approaches (e.g., Bischof et al. (2012)) combine and/or extend existing query languages to enable querying of different data formats. This category also includes such approaches, in which queries are translated from one language into another, as in (Progress Software Corporation, 2016). The UniQue approach applies both language integration principles: UQL extends the expressiveness of Selectors with XPath, which in turn is translated to an equivalent XPath / XQuery before executing the query.

To overcome common challenges of querying heterogeneous web content, Berger et al. (2006) propose a novel language called Xcerpt. Xcerpt is a versatile query language (Bry et al., 2005) that is capable of accessing web data in all formats, such as XML and RDF. MashQL (Jarrar and Dikaiakos, 2012) is another example of a completely new query language. Because of the originality of Xcerpt and MashQL, however, the query languages have not gained popularity among web mashup developers. Tsai et al. (2011) and YQL (Yahoo, 2016) present a proxy-based solution that uses a more mainstream query language as a basis, namely SQL. In these two approaches, an SQL-like query language is used to perform CRUD operations on heterogeneous web data sources and RESTful APIs, respectively. Our approach aims to minimize the learning effort and technologies required by developers, and thus takes full advantage of developers' existing knowledge on open W3C standards.

8 CONCLUSIONS

In this paper, we introduced a unified querying (UniQue) approach that provides a uniform and declarative query interface across heterogeneous web data sources. The proposed approach is based on the idea of data format and query language unification. Additionally, our approach leverages web mashup developers' prior knowledge on open W3C standards to enable broad adoption. We proposed Unified Query Language (UQL) that seamlessly extends the expressiveness of CSS Selectors with XPath expressions to query our unified data model. Both the UniQue approach and UQL were realized in our proxy-based implementation, which is made available under the terms of the MIT license at `https://mediatech.aalto.fi/publications/webservices/unique/`. The evaluation results from our case study indicate that UniQue can effectively streamline web querying, and show up to 51% (with compression) and 3% (without compression) reduction in generated network traffic compared to the current state-of-the-art approach.

ACKNOWLEDGEMENTS

The authors thank William Candillon and Dr. Ghislain Fourny at 28.io for their exceptional support with XQuery.

REFERENCES

28msec, 2016. 28msec. http://www.28.io/.

Akamai, 2016. Ion Web Performance Optimization | Akamai. https://www.akamai.com/us/en/solutions/products/web-performance/web-performance-optimization.jsp.

Bailey, J., Bry, F., Furche, T., Schaffert, S., 2005. Web and Semantic Web Query Languages: A Survey. In

First International Summer School 2005, LNCS 3564, pp. 35–133. Springer.

Berger, S., Bry, F., Furche, T., Linse, B., Schroeder, A., 2006. Beyond XML and RDF: The Versatile Web Query Language Xcerpt. In *Proceedings of the 15th International Conference on World Wide Web*, pp. 1053–1054. ACM.

Bischof, S., Decker, S., Krennwallner, T., Lopes, N., Polleres, A., 2012. Mapping between RDF and XML with XSPARQL. *Journal on Data Semantics*, vol. 1, no. 3, pp. 147–185. Springer.

Boyer, J.M., Bruchez, E., Klotz, L.L. Jr., Pemberton, S., Van den Bleeken, N., 2016. XForms 2.0 - W3C XForms Group Wiki (Public). http://www.w3.org/MarkUp/Forms/wiki/XForms_2.0.

Boyer, J., Gao, S., Malaika, S., Maximilien, M., Salz, R., Simeon, J., 2011. Experiences with JSON and XML Transformations. In *W3C Workshop on Data and Services Integration*.

Bray, T., 2014. The JavaScript Object Notation (JSON) Data Interchange Format - RFC 7159 (Proposed Standard). http://tools.ietf.org/html/rfc7159.

Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition) - W3C Recommendation. http://www.w3.org/TR/xml/.

Bry, F., Koch, C., Furche, T., Schaffert, S., Badea, L., Berger, S., 2005. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *International Journal on Semantic Web and Information Systems*, vol. 1, no. 2, pp. 1–21. IGI Global.

Çelik, T., Etemad, E.J., Glazman, D., Hickson, I., Linss, P., Williams, J., 2011. Selectors Level 3 - W3C Recommendation. http://www.w3.org/TR/selectors/.

Etemad, E.J., Atkins, T. Jr., 2016. Selectors Level 4 - W3C Editor's Draft. https://drafts.csswg.org/selectors/.

Florescu, D., Fourny, G., 2013. JSONiq: The History of a Query Language. *IEEE Internet Computing*, vol. 17, no. 5, pp. 86–90. IEEE.

Furche, T., Gottlob, G., Grasso, G., Schallhart, C., Sellers, A., 2013. OXPath: A Language for Scalable Data Extraction, Automation, and Crawling on the Deep Web. *The VLDB Journal*, vol. 22, no. 1, pp. 47–72. Springer.

Giribet, D., 2005. Merging XPath and URLs for Enhanced Web and Web Service Data Retrievals. In *Proceedings of the IADIS International Conference on Applied Computing*, pp. 27–33. IADIS.

Google, 2016. Data Saver - Google Chrome. https://developer.chrome.com/multidevice/data-compression.

Harth, A., Norton, B., Polleres, A., Sapkota, B., Speiser, S., Stadtmüller, S., Suominen, O., 2011. Towards Uniform Access to Web Data and Services. In *W3C Workshop on Data and Services Integration*.

Hausenblas, M., Wilde, E., Tennison, J., 2014. URI Fragment Identifiers for the text/csv Media Type – RFC 7111 (Informational). http://tools.ietf.org/html/rfc7111.

Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Doyle Navara, E., O'Connor, E., Pfeiffer, S., 2014. HTML5: A Vocabulary and Associated APIs for HTML and XHTML - W3C Recommendation. http://www.w3.org/TR/html5/.

Jarrar, M., Dikaiakos, M.D., 2012. A Query Formulation Language for the Data Web. *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 5, pp. 783–798. IEEE.

Kosek, J., Atkins, T. Jr., 2016. Non-Element Selectors Module Level 1 - W3C Editor's Draft. https://drafts.csswg.org/selectors-nonelement/.

MacLean, A., Carter, K., Lövstrand, L., Moran, T., 1990. User-Tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 175–182. ACM.

Pemberton, S., Austin, D., Axelsson, J., Çelik, T., Dominiak, D., Elenbaas H., Epperson, B., Ishikawa, M., Matsui, S., McCarron, S., Navarro, A., Peruvemba, S., Relyea, R., Schnitzenbaumer, S., Stark, P., 2002. XHTML^{TM} 1.0 The Extensible HyperText Markup Language (Second Edition): A Reformulation of HTML 4 in XML 1.0 - W3C Recommendation. http://www.w3.org/TR/xhtml1/.

Progress Software Corporation, 2016. DataDirect XQuery Product Architecture Overview. http://www.progress.com/products/data-integration-suite/xquery/xquery-product-architecture.

Robie, J., Chamberlin, D., Dyck, M., Snelson, J., 2014. XML Path Language (XPath) 3.0 - W3C Recommendation. http://www.w3.org/TR/xpath-30/.

Robie, J., Chamberlin, D., Dyck, M., Snelson, J., 2014. XQuery 3.0: An XML Query Language - W3C Recommendation. http://www.w3.org/TR/xquery-30/.

Shafranovich, Y., 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files - RFC 4180 (Informational). http://tools.ietf.org/html/rfc4180.

Tsai, C.-L., Chen, H.-W., Huang, J.-L., Hu, C.-L., 2011. Transmission Reduction between Mobile Phone Applications and RESTful APIs. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 445–450. ACM.

Van Roy, P., Haridi, S., 2004. *Concepts, Techniques, and Models of Computer Programming*, The MIT Press. Cambridge, Massachusetts, 1st edition.

Yahoo, 2016. Yahoo Query Language (YQL). http://developer.yahoo.com/yql/.