Ontop4theWeb: SPARQLing the Web On-the-fly

Konstantina Bereta
Dept. of Informatics
and Telecommunications, UoA
Email: konstantina.bereta@di.uoa.gr

George Papadakis
Dept. of Informatics
and Telecommunications, UoA
Email: gpapadis@di.uoa.gr

Manolis Koubarakis
Dept. of Informatics
and Telecommunications, UoA
Email: koubarak@di.uoa.gr

***Abstract*—Web data come in many different structures and formats that are not supported by Semantic Web tools. To leverage them, we propose a system, called Ontop4TheWeb, which allows for mapping Web data of various formats into virtual RDF triples and querying them on-the-fly without materializing them as RDF triples. We put our system into practice, querying with SPARQL diverse Web data sources that range from HTML tables to REST APIs, such as social media APIs. Our thorough experimental evaluation demonstrates the high efficiency of our approach, which goes beyond the current state-of-the-art in this area, in terms of both functionality and performance.**

I. Introduction

Querying Web data sources *on-the-fly* is an important task for several reasons: (i) Having full access to such data sources may involve a high economic cost (e.g., the price of subscribing to the entire Twitter stream). (ii) The constantly changing terms of use and the corresponding legislation complicates data crawling (e.g., the constraints defined by the recent EU General Data Protection Regulation[1]). (iii) The high frequency of updates (**Velocity**) makes it difficult for data consumers to synch with Web sources like social media applications. E.g., $\sim$6.000 tweets are posted per second in Twitter[2].

Moreover, querying *non-RDF* Web data on-the-fly *using SPARQL* has become a major issue, given that many Web data sources rely on REST APIs and HTML tables. Several solutions address this issue by extending standards, such as the SPARQL language [1], the SPARQL protocol [2], or the R2RML mapping language [3], to provide primitives for querying various kinds of Web data sources, such as APIs.

However, these works merely support relational data or specific file formats (e.g., XML, CSV). They also rely on custom SPARQL/R2RML extensions that hamper their adoption, while their combination with third-party added-value services is a very complicated procedure. Some of them also implement a caching mechanism such as [1] and [2], but do not fully exploit it, as shown in Section IV.

In this paper, we extend the traditional OBDA paradigm with support for non-relational, Web data. We propose a novel OBDA-based system architecture that is able to query Web APIs and other sources on-the-fly using SPARQL, without extending standards such as SPARQL and R2RML. Our main contributions are the following:

- We introduce an extension of the OBDA paradigm to support data sources not materialised in relational native databases.
- We implement our approach in the system Ontop4theWeb, an OBDA-based system for posing SPARQL queries on top of non-RDF Web data on-the-fly. To achieve this, virtual table operators are embedded in the SQL queries that are included in R2RML mappings. These mappings specify which part and source of Web data will be fetched and how they will be mapped to virtual RDF terms. Combining these mappings with an ontology allows for returning the virtual relational data that are involved in the query as RDF results.
- We perform a qualitative and quantitative, experimental, comparison of our system with the state-of-the-art system described in [1], and we show that Ontop4theWeb provides more functionality, is more efficient and does not deviate from the standards.

The rest of the paper is organized as follows: Section II presents background and related work. Section III describes the implementation and Section IV presents our experimental evaluation and comparison to the state-of-the-art. Section V concludes the paper along with directions for future work.

II. Background and Related Work

OBDA systems [4] are primarily useful in cases where users store their data in relational databases, but do not want to materialize them as RDF triples, particularly when these databases are large or/and get frequently updated, due to Velocity [5]. As a result, many OBDA and RDB2RDF systems have been developed in the recent years, such as Ontop [6], Ultrawrap [7], Morph [8], Sparqlify[3], and Oracle Spatial and Graph[4]. Most of them support the R2RML mapping language or similar native languages.

Closer to our work is the *SERVICE-to-API* system [1], which proposes an extension of SPARQL that enables users to combine the responses of JSON APIs with results from the evaluation of standard triple patterns. We deviate from this approach in that: (i) we do not extend SPARQL syntax, (ii) we allow users to query APIs using standard SPARQL triple patterns directly, without having to combine them with stored RDF data, (iii) we provide a general approach that is not limited to JSON APIs, and (iv) we produce significantly

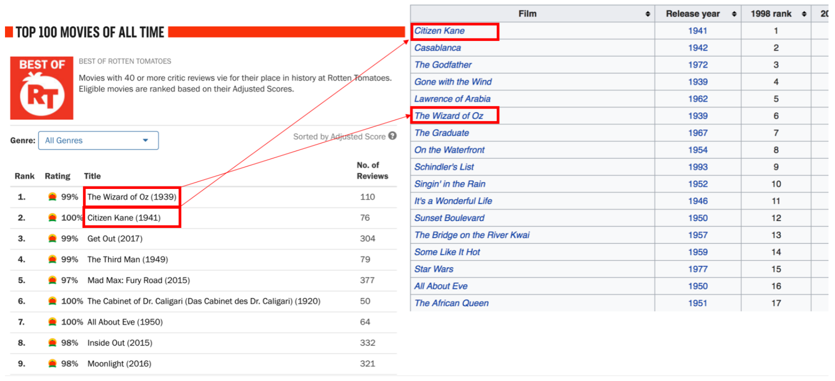
This work was partially funded by the EU H2020 project ExtremeEarth (Grant agreement No. 825258).

[1]See https://eugdpr.org for more details.

[2]http://www.internetlivestats.com/twitter-statistics/

[3]http://aksw.org/Projects/Sparqlify.html

[4]http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.html



TOP 100 MOVIES OF ALL TIME

BEST OF ROTTEN TOMATOES

Movies with 40 or more critic reviews vie for their place in history at Rotten Tomatoes. Eligible movies are ranked based on their Adjusted Scores.

Genre: All Genres

Sorted by Adjusted Score

Rank	Rating	Title	No. of Reviews
1.	99%	The Wizard of Oz (1939)	110
2.	100%	Citizen Kane (1941)	76
3.	99%	Get Out (2017)	304
4.	99%	The Third Man (1949)	79
5.	97%	Mad Max: Fury Road (2015)	377
6.	100%	The Cabinet of Dr. Caligari (Das Cabinet des Dr. Caligari) (1920)	50
7.	100%	All About Eve (1950)	64
8.	98%	Inside Out (2015)	332
9.	98%	Moonlight (2016)	321

Film	Release year	1998 rank	20
Citizen Kane	1941	1	
Casablanca	1942	2	
The Godfather	1972	3	
Gone with the Wind	1939	4	
Lawrence of Arabia	1962	5	
The Wizard of Oz	1939	6	
The Graduate	1967	7	
On the Waterfront	1954	8	
Schindler's List	1993	9	
Singin' in the Rain	1952	10	
It's a Wonderful Life	1946	11	
Sunset Boulevard	1950	12	
The Bridge on the River Kwai	1957	13	
Some Like It Hot	1959	14	
Star Wars	1977	15	
All About Eve	1950	16	
The African Queen	1951	17	

Fig. 1: Tables with 100 movies from Rotten Tomatoes (left) and Wikipedia (right).

fewer API calls, which translates to improved performance, as we explain in detail in Section IV.

Another related approach is based on the development of SPARQL wrappers for Web APIs [2]. To this end, it extends HTTP requests to SPARQL endpoints, including arguments that are used to retrieve a fragment of the data that can be accessed via the Web API. This fragment is converted into RDF and stored using an in-memory triple store. We deviate from this approach in that: (i) no data are materialized into RDF, as the query is converted on-the-fly using mappings, (ii) our approach can be adapted to a different schema simply by modifying the mapping file, without requiring a change in the system code, unlike [2], and (iii) the translation of the original SPARQL query is completely transparent to the end-user, whereas [2] requires the end user to be aware of the Web API documentation so as to specify the fragment of the Web API he wants to access.

III. System Architecture and Implementation

In this section we describe the methdology and the implementation of our system, Ontop4TheWeb. Its architecture is shown in Figure 2 and it builds on the following components:

MadIS. The back-end relies on *MadIS*[5] [9], a relational database system based on *SQLite* that can be extended with user-defined row, aggregate, or virtual table operators. MadIS exploits the APSW SQLite wrapper, which provides a Python interface for implementing such operators in an extensible way. Using APSW, we define our own virtual table operators and populate them with data retrieved from the Web. To query the retrieved data, we use *MadQL*, the MadIS implementation of an extended-SQL language, which supports user defined functions embedded in SQL queries.

We extended MadIS to support the caching mechanism that is described below as well as the virtual table operators that lie at the core of our approach. In fact, we define a *virtual table operator* for each kind of data source as: $\mathbf{VT} ::= \mathbf{vtable}(\mathbf{args}[\ldots, \mathbf{f}])$, where the vector $args$ denotes the arguments that are given as input to the virtual table operator, while f is optional, denoting the *cache update rate*. In relational algebra, a virtual table ($vtable$) is handled as a relational table. Thus, any mapping language that allows for SQL queries in mappings is compatible (e.g., R2RML).

Instead of extending MadIS, we could implement the same virtual table operators in C, extending SQLite directly, but this would be less user-friendly and less re-usable than the plug-and-play MadIS Python operators. It would also undermine the modularity and extensibility of Ontop4theWeb.

Caching. To improve performance, each virtual table can optionally use a cache. The cache feature is useful in cases where: (i) not all data sources get updated with the same frequency, (ii) some data sources might not be accessible at the next query time (e.g., due to API limitations), or (iii) a minimal query execution time is required, due to a large number of queries, i.e., the frequency of queries is much higher than the update frequency of data sources. To support these cases, f indicates the length of the time window (in milliseconds), during which the retrieved data are temporarily stored. If the virtual table operator with the *same* input parameters ($args$) is invoked twice (or more) before this time window ends, the cached data will be used, improving query time. If the query is repeated after the end of the time window, the fresh data is fetched from the data source and gets stored in the system. If f gets a negative or null value, nothing is stored and the virtual table operator fetches fresh data every time it is invoked. This is supported by storing metadata about when and where data resulting from a virtual table signature was stored last time.

Ontop-spatial[6] [10] is the geospatial extension of the OBDA system Ontop [6]. As an OBDA system, Ontop-spatial connects only to populated and materialized databases, using their data for optimization, *before* querying them. Instead, Ontop4theWeb retrieves the data to be queried only *after* the user fires a query, creating a virtual table on-the-fly without any prior knowledge of the data. We have extended the MadIS JDBC connector so that it complies with Ontop-spatial.

Third-party applications. These are external *micro-services* that can be invoked by a virtual table operator in MadIS. For example, a virtual table operator is able to identify the polarity of a tweet by calling a micro-service that implements a Sentiment Analysis classifier (see below). This feature enables Ontop4theWeb to perform data analysis tasks without facing

[5]http://madgik.github.io/madis

[6]http://ontop-spatial.di.uoa.gr/

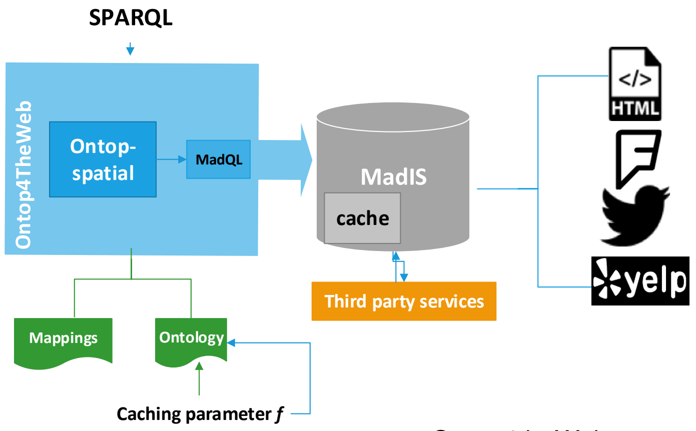


Fig. 2: System architecture for Ontop4theWeb.

any compatibility issues between the corresponding component and the `vtable` operator: the server can be written in any language, but the client can still use it as a service.

IV. Experimental evaluation

In this section we perform a qualitative and a quantitative comparison of our work and state-of-the-art approach described in [1].

1) Qualitative comparison: The *SERVICE-to-API* system [1] enriches RDF data with data from external sources, such as REST APIs. Thus, its query language requires at least one triple pattern to be evaluated in the RDF repository and its variables to be bound to values that populate their URI templates. Every variable binding yields a separate API call. A cache mechanism aims to minimize the API calls. An example is presented in Listing 1. The value of keyword $SERVICE$ creates a URI template for each one of the values bound to the variable l, which is used in the query's triple pattern. In this case, a call to the Yelp API is produced for each binding of the variable l, returning a JSON file. This JSON file is parsed according to the JSON pattern included in the query, which bounds the variables i, $name$ and $rating$ to the values of the respective attributes of the JSON file.

Listing 1: Get Yelp businesses with a SERVICE-to-API query

```
SELECT   ?i ?name  WHERE {
?x <http://www.w3.org/2000/01/rdf-schema#label> ?l .
SERVICE <https://api.yelp.com/v3/businesses/{l}>{
( $.[\"id\"], $.[\"name\"]) AS (?id, ?name)}}
```

In this context, there are two major *qualitative* differences between Ontop4theWeb and SERVICE-to-API [1]:

1) The query language. For SERVICE-to-API, the JSON attributes are directly bound to variables by parsing the JSON response, as instructed by the JSON patterns included in the query. As a result, the users need to know the documentation of the API in order to identify the information they need. Only in this way are they able to combine API data with the RDF data in the triplestore, formulating accurate queries that extend SPARQL with JSON patterns [1]. In contrast, Ontop4theWeb creates virtual semantic graphs on top of APIs using mappings, thus allowing users to pose standard SPARQL queries as if the contents of the APIs were transformed into RDF. The trade-off for not having to convert, materialise and store the data into an RDF store is the use of mappings. For any virtual Ontop4theWeb RDF repository, a mapping file should be provided. Writing these mappings can be an overhead, but they need to be written once, unless the schema changes. Note, though, that the mapping language R2RML is W3C standard, just as the SPARQL query language, ensuring compatibility with applications built on top of SPARQL. Also, materialisation is not avoided in SERVICE-to-API, as the data should be partially stored in a triplestore.

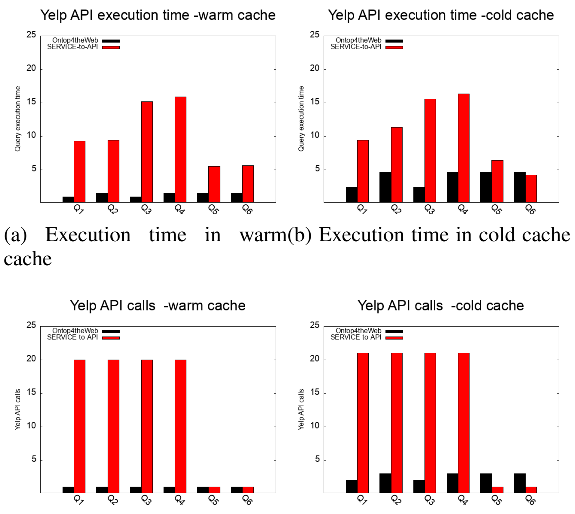
2) Both systems use a caching mechanism, but every API call in Ontop4theWeb retrieves an entire virtual table, which is mapped to a virtual RDF graph. Instead, SERVICE-to-API merely retrieves one entry of this table per API call, which has a significant impact on time efficiency, as explained below.

2) Quantitative comparison: To compare the performance of the two systems, we use data retrieved from the REST API of Yelp, the only data source for which both systems offer the same functionality However, the findings of this experiment are representative of the general behaviour of the two systems.

For Ontop4theWeb, we implemented a virtual table operator of Yelp and pose the SPARQL query Q1 to retrieve business names and ids from the Yelp API. For SERVICE-to-API, we used the original implementation of SERVICE-to-API [1]. For the corresponding SERVICE-to-API query, we stored data about burger joints in Chicago in an RDF repository, because this system does not support API calls without triple patterns included in the query. Then, we used the SERVICE keyword to join them with their names and IDs that are retrieved from the REST API of Yelp. In SERVICE-to-API query Q2, we want to retrieve the names of businesses and their ids from the Yelp API, but it returns many false positives that were produced because the values that are bound to the variables of the query do not get joined, as in the case when the names are materialised in SERVICE-to-API query Q1.

SPARQL Query Q2 contains one more triple pattern (i.e., it also retrieves the rating of businesses). There are different ways to express this query in the SERVICE-to-API, depending on the configuration of the repository. The closest definition seems to be the SERVICE-to-API query Q3. However, the query actually returned the Cartesian product of all different burger businesses and all different rating values.

To compare the two systems on an equal basis, we consider



(a) Execution time in warm cache (b) Execution time in cold cache

(c) API calls in warm cache (d) API calls in cold cache

Fig. 3: Execution times for Yelp queries.

all different variations of the standard SPARQL query Q2. Having at least one triple for each entity stored, we retrieve only the missing values using the SERVICE-to-API query Q4. In this way, SERVICE-to-API returns the correct results, since the underlying triple store is forced to perform a JOIN between the materialized and the values that are returned from the API, instead of a Cartesian product. The trade-off, on the other hand, is that *SERVICE-to-API cannot pose a query to retrieve the results directly through the API, as some form of materialization needs to be performed in order to retrieve correct results*. In SERVICE-to-API query Q3, we stored the names in the triple store and retrieved the ratings and IDs from the API producing a Cartesian product $N \times S$. SERVICE-to-API query Q6 differs from query Q5 only in that it uses the `BIND` operator instead of triple pattern (i.e., instead of storing the respective triple in a triple store). This allows for executing a materialised-nothing query, such as the one that is performed in Ontop4theWeb query. The results of this query are the same as those of SERVICE-to-API query Q5.

Response time. Figures 3(a) and (b) depict the query execution times of the above queries, while Figures 3(c) and (d) illustrate the number of API calls invoked. In both cases, we consider warm and cold caches (on the left and right, respectively). We observe that Ontop4theWeb is three times faster than SERVICE-to-API because Ontop4theWeb retrieves a set of tuples for each API call while SERVICE-to-API retrieves one entry for each API call, yielding more API calls. Ontop4theWeb by design also benefits more from caching than the system in comparison. We cache the entire table for each API call, while SERVICE-to-API performs an API call for each tuple, which means that only one tuple is cached each time.

Accuracy. Table I shows the accuracy of the results returned by the two systems. Ontop4TheWeb consistently

	SPARQL	SERVICE-to-API		SPARQL	SERVICE-to-API			
	Q1	Q1	Q2	Q2	Q3	Q4	Q5	Q6
Precision	1.00	1.00	0.05	1.00	0.05	1.00	0.02	0.02
Recall	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
F-score	1.00	1.00	0.09	1.00	0.10	1.00	0.03	0.03
Accuracy	1.00	1.00	0.05	1.00	0.05	1.00	0.02	0.02

TABLE I: Ontop4theWeb vs SERVICE-to-API effectiveness.

achieves perfect accuracy and F1-score[7], unlike SERVICE-to-API, which produces false positives that reduce the system's precision and accuracy and, inevitably, its F1-score. The reason is that it returns the Cartesian product of the bindings of all variables involved in an API call.

V. Conclusions

This paper presents our extension to the OBDA paradigm implemented in the system Ontop4theWeb, an open-source system for querying Web data on-the-fly using SPARQL. Ontop4theWeb extends SQL with virtual table operators, embeds them into mappings and makes an OBDA system compliant with them. Our extensive experimental evaluation verified that Ontop4theWeb goes beyond the state of the art with respect to functionality and performance. In the future, we will apply our system to more applications that include data analysis tasks and make the results available as virtual RDF triples on-the-fly.

References

[1] M. Mosser, F. Pieressa, J. L. Reutter, and A. Soto, "Querying apis with SPARQL: language and worst-case optimal algorithms," in *ESWC*, 2018, pp. 639–654.

[2] F. Michel, C. Faron-Zucker, and F. Gandon, "Sparql micro-services: Lightweight integration of web apis and linked data," in *Workshop on Linked Data on the Web co-located with WWW*, 2018.

[3] A. Chortaras and G. Stamou, "Mapping diverse data to RDF in practice," in *ISWC*, 2018, pp. 441–457.

[4] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati, "Linking data to ontologies," in *Journal on Data Semantics*, vol. 10, 2008, pp. 133–173.

[5] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi, and D. F. Savo, "The MASTRO System for Ontology-based Data Access," *Semant. Web journal*, vol. 2, no. 1, 2011.

[6] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao, "Ontop: Answering SPARQL queries over relational databases," *Semantic Web*, vol. 8, no. 3, pp. 471–487, 2017.

[7] J. F. Sequeda and D. P. Miranker, "Ultrawrap: SPARQL execution on relational data," *J. Web Sem.*, vol. 22, pp. 19–39, 2013.

[8] F. Priyatna, Ó. Corcho, and J. F. Sequeda, "Formalisation and experiences of R2RML-based SPARQL to SQL query translation using Morph," in *WWW14*, pp. 479–490.

[9] Y. Chronis, Y. Foufoulas, V. Nikolopoulos, and et al, "A Relational Approach to Complex Dataflows," in *EDBT/ICDT Workshops*, 2016.

[10] K. Bereta and M. Koubarakis, "Ontop of Geospatial Databases," in *Proceedings of the 15th International Semantic Web Conference*, 2016.

[7]https://github.com/ConstantB/Ontop4TheWeb/tree/experiments