ObjectRunner: Lightweight, Targeted Extraction and Querying of Structured Web Data

Talel Abdessalem Bogdan Cautis Nora Derouiche

Télécom ParisTech - CNRS LTCI, Paris, France

{firstname.lastname@telecom-paristech.fr}

ABSTRACT

We present in this paper ObjectRunner, a system for extracting, integrating and querying structured data from the Web. Our system harvests real-world items from template-based HTML pages (the so-called structured Web). It illustrates a two-phase querying of the Web, in which an intentional description of the targeted data is first provided, in a flexible and widely applicable manner. ObjectRunner follows then a lightweight, best-effort approach, leveraging both the input description and the source structure. This process is domain-independent, in the sense that it applies to any relation, either flat or nested, describing real-world items. We advocate via our prototype that fully automatic extraction and integration of structured data can be done fast and effectively, when the redundancy of the Web meets knowledge over the to-be-extracted data. We present the technical details and the overall platform through several application scenarios on real-life Web sources.

1. INTRODUCTION

Extracting structured information from the ocean of Web data is one of the key challenges in data management research today, and of foremost importance in the larger effort to bring more semantics to the Web. In short, its aim is to map as accurately as possible Web page content to relational-style tables.

Also, we witness in recent years a steady growth of the so-called structured Web. This represents documents (Web pages) that are data-centric, presenting structured content, complex objects. Such schematized pages are often generated dynamically by means of formatting templates over a database, possibly using user input via forms (in hidden Web pages). Moreover, there is also strong recent development of the collaborative Web, representing efforts to build rich repositories of user-generated structured content.

We present in this paper ObjectRunner, a system that aims to extract complex data from the structured Web.

Much work has been done recently on techniques for structured Web information extraction. These are pages that (i) share a common schema for the information they exhibit, and (ii) share a common template to encode this information (for a survey, see [10]). The techniques that apply to schematized Web sources are generally called *wrapper inference* techniques, and have been extensively studied in the literature, ranging from supervised (hard-coded) wrappers to fully unsupervised ones. At the end of the spectrum, there have been several proposals for automatically wrapping structured Web sources, such as [1, 6, 16]. Their approach is usually generic, in the sense that only the pages' regularity is exploited, be it at the level of HTML encoding or of the visual rendering of pages. The extracted data is used to populate a relational-style table, a priori without any knowledge over its content. Adding semantics can then be done either by manual labeling or even by automatic post-processing (a non-trivial problem in its own). In practice, this approach suffers from two significant shortcomings:

- only part of the resulting data may be of real interest for a given user/application, hence effort may be spent on valueless information,
- with no insight over its content, data resulting from the extraction process may mix values corresponding to distinct attributes of the implicit schema, making the subsequent labeling phase tedious and error-prone.

The usability of the collected data is therefore often restricted in real-life scenarios.

We address these shortcomings with the ObjectRunner project, based on a paradigm of two-phase querying of the Web that leverages both the content and structure of the pages. ObjectRunner is attacking the wrapping problem from the angle of users looking for a certain kind of information on the Web. More precisely, it starts from an intentional description of the targeted data, denoted *Structured Objet Description* (in short *SOD*), which is provided by users in a minimal-effort, flexible manner. The interest of having such a description is twofold: it allows to improve the accuracy of the extraction process, in many cases quite significantly, and it makes this process more efficient (lightweight) by enabling the elimination of unnecessary computations.

System overview. A high-level view of the demonstrated system is illustrated in Figure 1 (it will then be discussed in more detail in the following sections). Users are provided with widely applicable tools that allow them to specify via an SOD (to be formally defined shortly) what must be obtained from Web pages, in particular what atomic types (i.e., simple entities) are involved in the intentional description and how (e.g., occurrence constraints, nesting, value joins). Techniques to handle both existing (built-in) and new atomic types efficiently are provided. Starting from a corpus of Web sources, where each source represents a set of pages with common (implicit) schema and structure, it then builds an extraction template (wrapper) and harvests the objects - possible instances of the given SOD - from these pages. Both structured data and textual information related to it are then indexed in the ObjectRunner repository. In the interrogation phase, users may select one or several SODs. This triggers the generation of a

Proceedings of the VLDB Endowment, Vol. 3, No. 2

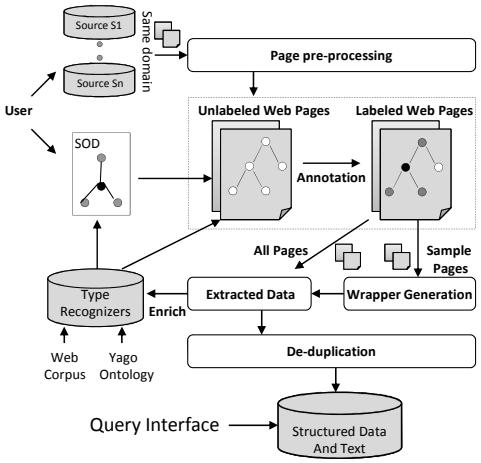

Figure 1: Architecture of ObjectRunner

query interface in the style of Query-By-Example, in which both structured and unstructured data (keywords) might be considered. Query results are sorted, among other criteria, based on confidence scores from the extraction process.

The demonstration will focus mainly on the technical aspects of the extraction and on the features of the specification and interrogation interfaces. Beyond these aspects, there are other exciting research problems we are currently investigating. For instance, how could one discover, process and index in a scalable and effective manner large corpora of structured Web pages, as potential sources for `ObjectRunner`? Or how could one select the most relevant sources for a given SOD? A discussion of these issues goes beyond the scope of this demonstration proposal, whose main purpose is to advocate the avantages of the two-phase querying approach.

Our experiments show that by (i) having an explicit target for the extraction process, and (ii) using diverse and rich enough sources, this approach turns out to be highly effective in practice. Moreover, preliminary results hint that a fully automatic solution for querying the structured, non-hidden Web - including aspects such as source indexing and selection - might be within reach, based on carefully designed heuristics and the redundancy of the Web.

2. PRELIMINARIES

We introduce in this section the necessary terminology and technical background. We suppose that the user has collected a number of structured Web sources $\{S_1, \ldots, S_n\}$, where each source represents a set of `HTML` pages that describe real-world objects (e.g., concerts, real-estate ads, books, etc). Our running example refers to `concert` objects, which can be seen as triples date-address-artist. We illustrate in Figure 2 four fragments of template-based pages that describe such information. We start by defining the typing formalism by which one can specify what data should be extracted from the `HTML` pages. We then discuss the extraction problem.

Types. We consider a set of *entity types*, where each such type represents an *atomic* piece of information, expressed as a string of tokens (words or `HTML` tags). Each entity type t_i has an associated *recognizer* r_i which can be simply viewed as a regular expression. In practice, as in the demonstration, we will distinguish three kinds of recognizers: (i) user-defined regular expressions, (ii) system predefined ones (e.g., addresses, dates, phone numbers, etc), and (iii) open, dictionary-based ones (denoted in our interface *isInstanceOf* recognizers; see Figure 3). We discuss more the recognizer choices and implementation in the next section.

Based on entity types, we define recursively complex types. A *set type* is a pair $t = [\{t_i\}, m_i]$ where $\{t_i\}$ denotes a set of instances of type t_i (atomic or not) and m_i denotes a multiplicity constraint that specifies restrictions on the number of t_i instances in t: $n - m$ for at least n and at most m, $*$ for zero or more, $+$ for one or more, ? for zero or one. A *tuple type* denotes an unordered collection of set or tuple types. A *disjunction type* denotes a pair of mutually exclusive types.

SODs. A Structured Object Description denotes any complex type, possibly complemented by additional restrictions in the form of value, textual or disambiguation rules. For instance, these would allow one to say that a certain entity type has to cover the entire textual content of an `HTML` node or a textual region delimited by consecutive `HTML` tags. Or to require that two *date* types have to be in a certain order relationship or that a particular address has to be in a certain range of geographical coordinates. For brevity, these details are omitted in the model described here.

An *instance* of an entity type t_i is any string that is valid w.r.t. the recognizer r_i. Then, an instance of an SOD is defined straightforwardly in a bottom-up manner.

For example, `concert` objects could be specified by an SOD as a tuple type composed of three entity types: one for the *address*, one for the *date* and one for the *artist name*. The first two would be associated to predefined recognizers (for adresses and dates respectively), while the last one would have an *isInstanceOf* recognizer.

Extraction templates. For a given SOD s and source S_i, a *template* τ w.r.t. s and S_i describes how instances of s can be extracted from S_i pages. More precisely,

- for each set type $t = [\{t_i\}, m_i]$ appearing in s, τ defines a *separator* string sep^t; it denotes that consecutive instances of t_i will be separated by this string.
- for each tuple type $t = \{t_1, \ldots, t_k\}$, τ defines a total order over the collection of types and a sequence of $k+1$ *separator* strings $sep^t_1, \ldots, sep^t_{k+1}$; this denotes that the k instances of the k types forming t, in the specified order, will be delimited by these separators.

We are now ready to describe the extraction problem we consider. For a given SOD s and a set of sources $\{S_1, \ldots, S_n\}$,

1. **set up** type recognizers for all the entity types in s,
2. for each source S_i,
   (a) **find** and **annotate** entity type instances in pages,
   (b) **select** a sample set of pages,
   (c) **infer** a template $\tau_i(s, S_i)$ based on the sample,
   (d) use τ_i to **extract** all the instances of s from S_i,
3. **refine** the recognizers based on the extracted objects.

3. IMPLEMENTATION OVERVIEW

We provide in this section a more detailed description of the system's internal structure in terms of composing parts and implementation approaches. We also illustrate how it operates through an example. The underlying principle of `ObjectRunner` is that, given the redundancy of Web data, solutions that are computationally less expensive, yet have high precision and satisfactory recall, should be favored in most aspects of the system. Though this means that some sources may be poorly handled, it is highly likely in practice that their data can be found elsewhere as well and, overall, the performance speed-up is deemed much more important.

Broadly, the extraction process is done in two stages: (1) automatic annotation, which consists in recognizing instances of the input SOD's entity types in page content, and (2) extraction template construction, using the semantic annotations from the previous stage and the regularity of pages.

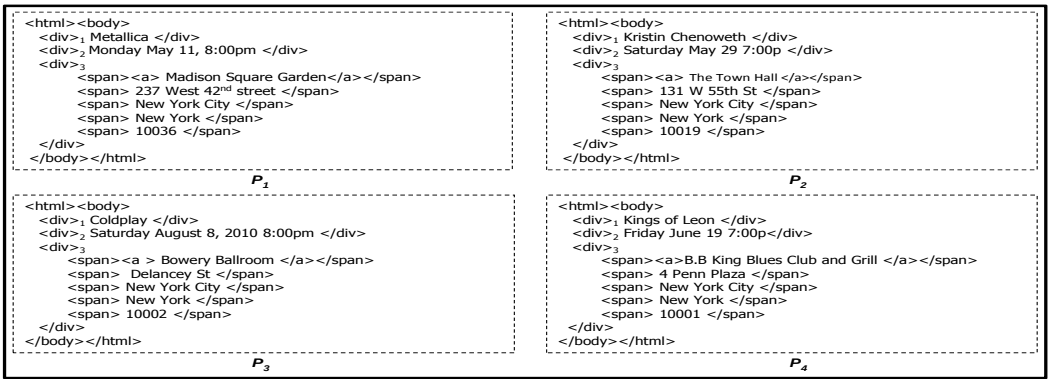

Figure 2: Sample pages

Page pre-processing. Pre-processing is necessary in order to clean `HTML` pages, e.g., to remove header details, scripts, styles, comments, images, hidden tags, space, tag properties, empty tags, etc. In this step, the open source software JTidy is used to transform HTML documents to XML documents. Beyond page cleaning, we also apply on the collection of pages of each source a radical simplification to their "central" segment, the one which likely displays the main content of the page. For that, we rely on an algorithm that uses page segmentation (in the style of VIPS [2]) and the recognizers of the input SOD to chose the best candidate segment. For instance, the simplified pages in our example were obtained from the site `http://upcoming.yahoo.com/`.

Type recognizers. Importantly, in our application, type recognizers are never assumed to be entirely precise nor complete. This is inherent in the Web context, where different representation formats might be used for even the most common types of data. We only discuss here how *isInstanceOf* types are handled. Intuitively, these are types for which only a class name can be provided, without direct means to recognize instances thereof. This could be the case for the *Artist* entity type. When such a type is input by the user, `ObjectRunner` seamlessly constructs on the fly a dictionary-based recognizer for it. This can be done by querying the YAGO ontology [14], a vaste knowledge base built from Wikipedia and Wordnet (Yago has more than 2 million entities and 20 million facts). Despite its richness, useful entity instances may not be found simply by exploiting Yago's *isInstanceOf* relations. For example, `Metallica` is not an instance of the `Artist` class. This is why we look at a semantic neighborhood instead: e.g., `Metallica` is an instance of the `Band` class, which is semantically close to the `Artist` one. For our purposes, we adapted Yago in order to access such data with little overhead.

Alternatively, users can choose to look for instances directly on the Web, by applying Hearst patterns [8] on a corpus of Web pages that is pre-processed for this purpose. Other kinds of recognizers, e.g., based on Datalog-style rules or conditional-graphical models could be plugged in `ObjectRunner`. We are currently studying the overhead they might introduce in the system performance.

Annotation and page sample selection. No assumptions are made on the source pages. They may not be relevant for the input SODs, as they may even not be structured (template-based). The setting of our entity recognition sub-problem is the following: a certain number (typically small in practice) of entity types $t_1, \ldots, t_n$ have to be matched with a collection of pages (what we call a source). If done naively, this step could dominate the extraction costs, since we deal with a potentially large database of entity instances. Our approach here starts from the observation that only a subset of these pages have to be annotated, and from the annotated ones only a further subset (approx. 20 pages) are used as sample in the next stage, for template construction. We use selectivity estimates, both at the level of types and at the one of type instances, and look for entity matches in a greedy manner, starting from types with likely few witness pages and instances (see Algorithm 1). At each step, we continue the matching process only on the "richest" pages. We also take advantage of the inverted index, in the case of dictionary-based recognizers. During this loop, the source could be discarded if unsatisfactory annotation levels are obtained.

Algorithm 1 annotatePages

1: **input:** parameters (e.g., sample size k), source S_i, SOD s
2: sample set $S := S_i$
3: order entity types in s by selectivity estimate
4: **for all** entity types t in s **do**
5: look for matches of t in S and annotate
6: for $S' \sqsubseteq S$ top annotated pages, make $S := S'$
7: **end for**
8: **return** sample as most annotated k pages in S

Wrapper generation. This is the core component of the system. For each source S_i, its output is the extraction template τ_i corresponding to the input SOD s. We adopt in `ObjectRunner` an approach that is similar in style to the `ExAlg` algorithm of [1]. Like `ExAlg`, a template is inferred from a sample of source pages based on *occurrence vectors* for page tokens and *equivalence classes* defined by them (an equivalence class is a set of tokens having the same frequency of occurrences in each input page and a unique role). But how roles and equivalence classes are computed distinguishes our approach from [1]. First, we use annotations as an additional criterion for distinguishing token roles. This is an obvious strategy in our context but, since annotations are not complete and can be conflicting over the set of pages, has to be applied cautiously. In fact, we observe that it is the combination of equivalence class structure and annotations that yields the best results. Algorithm 2 sketches how token roles are differentiated.

Algorithm 2 diffTokens

1: differentiate roles using `HTML` features
2: **repeat**
3: **repeat**
4: find equivalence classes (EQs)
5: handle invalid EQs
6: diff. roles using EQs + non-conflicting annotations
7: **until** fixpoint
8: differentiate roles using EQs + conflicting annotations
9: **until** fixpoint

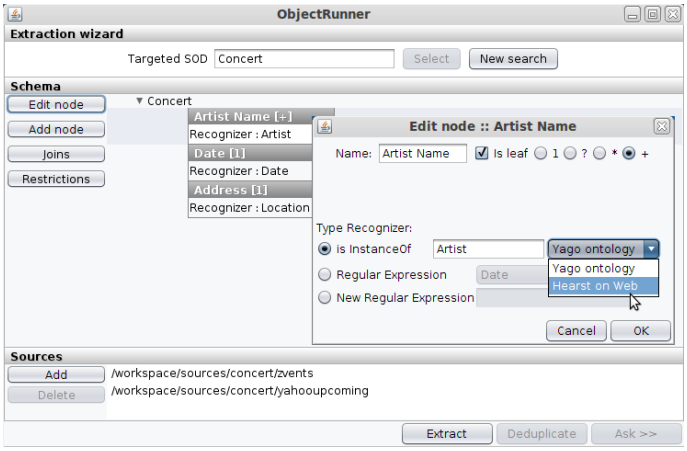

Figure 3: Extraction interface

Second, besides annotations, the SOD itself fulfills a double role during the wrapper generation step, as it allows us to: (i) stop the process as soon as we can conclude that the target SOD cannot be met (this might be the case, as the annotations alone do not guarantee success), and (ii) accept approximate equivalence classes outside the ones that might represent to-be-extracted instances.

On the sample pages of our example, if annotations are taken into account, we can detect that the `<div>` tag occurrences denoted $\texttt{div}_1$, $\texttt{div}_2$ and $\texttt{div}_3$ have different roles. By that, we can correctly determine how to extract the three components, artist, date and address. This would not be possible if only their positions in the `HTML` tree and in equivalence classes would be taken into account, as the three `<div>` occurrences would have the same *role*.

We give below the extraction template that would be inferred in our example:

```
<html><body>
    <div type="Artist"> * </div>
    <div type="Date"> * </div>
    <div type="Address">
        <span><a> * </a></span>
        <span> * </span>
        <span> * </span>
        <span> * </span>
        <span> * </span>
    </div>
</html></body>
```

4. DEMO SCENARIO

Our demonstration will focus on the desktop tools for extraction, interrogation and result browsing. First, as illustrated in the screen capture of Figure 3, users will be able define SODs. For that, they can either use existing SODs or types, or specify new types along with means to recognize them (e.g., using the YAGO ontology). Demo visitors will be able to try online this tool, choosing also the real-life sources to be used. In the query interface (Figure 4), users can choose which SODs will be used to query the Web sources. A QBE-style interface allows one to specify value restrictions, joins across SODs and keywords restrictions referring to the objects' source pages. Also, the sources that are to be queried can be chosen at this stage.

5. RELATED WORK

The existing works in Web data extraction can be classified according to their automation degree (for a survey, see [10]). The manual approaches extract only the data that the user marks explicitly, using either wrapper programming languages or visual platforms to construct extraction programs, like Lixto [7]. Supervised approaches use learning techniques, called *wrapper induction*, to

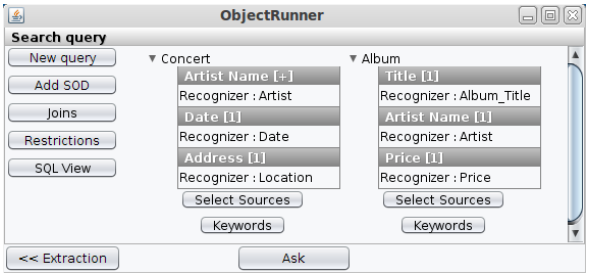

Figure 4: Query interface

learn extraction rules from manually labeled pages (XWrap [11]). Semi-supervised approaches (e.g., OLERA [3], Thresher [9]) arrive to reduce human intervention by acquiring a rough example from the user. Some semi-automatic approaches (such as IEPAD [4]) do not require labeled pages, but find extraction patterns according to extraction rules chosen by the user. The unsupervised approaches identify the to-be-extracted data using the regularity of the pages. One important issue is how to distinguish the role of each page component `(token)`, which could be either a piece of data or part of the encoding template. Some, as a simplifying assumption, consider that every `HTML` tag is generated by the template (as in DeLa [15], DEPTA [16]). RoadRunner [6], which uses an approach based on grammar inference, also assumes that every `HTML` tag is generated by the template, but other string tokens could be considered as part of the template as well. In comparison, ExAlg [1] makes more flexible assumptions, as the template tokens are those corresponding to frequently occurring equivalence class, and it can handle optional and alternative parts of pages. The TurboSyncer [5] system can incorporate many sources, using existing results to improve future extractions. Other works explore the mutual benefit of annotation and extraction, learning wrappers based on labeled pages [17, 12] or domain knowledge for query result records [13].

6. REFERENCES

[1] A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *SIGMOD Conference*, 2003.

[2] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma. Extracting content structure for web pages based on visual representation. In *APWeb*, 2003.

[3] C.-H. Chang and S.-C. Kuo. OLERA: Semisupervised web-data extraction with visual support. *IEEE Intelligent Systems*, 2004.

[4] C.-H. Chang and S.-C. Lui. IEPAD: information extraction based on pattern discovery. In *WWW*, 2001.

[5] S.-L. Chuang, K. C.-C. Chang, and C. Zhai. Context-aware wrapping: Synchronized data extraction. In *VLDB*, 2007.

[6] V. Crescenzi, G. Mecca, and P. Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In *VLDB*, 2001.

[7] G. Gottlob, C. Koch, R. Baumgartner, M. Herzog, and S. Flesca. The Lixto data extraction project - back and forth between theory and practice. In *PODS*, 2004.

[8] M. A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *COLING*, 1992.

[9] A. Hogue and D. R. Karger. Thresher: automating the unwrapping of semantic content from the world wide web. In *WWW*, 2005.

[10] M. Kayed and K. F. Shaalan. A survey of web information extraction systems. *IEEE TKDE*, 2006.

[11] L. Liu, C. Pu, and W. Han. XWrap: An XML-enabled wrapper construction system for web information sources. In *ICDE*, 2000.

[12] P. Senellart, A. Mittal, D. Muschick, R. Gilleron, and M. Tommasi. Automatic wrapper induction from hidden-web sources with domain knowledge. In *WIDM*, 2008.

[13] W. Su, J. Wang, and F. H. Lochovsky. ODE: Ontology-assisted data extraction. *ACM Trans. Database Syst.*, 34(2), 2009.

[14] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: a core of semantic knowledge. In *WWW*, 2007.

[15] J. Wang and F. H. Lochovsky. Data extraction and label assignment for web databases. In *WWW*, 2003.

[16] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *WWW*, 2005.

[17] J. Zhu, Z. Nie, J.-R. Wen, B. Zhang, and W.-Y. Ma. Simultaneous record detection and attribute labeling in web data extraction. In *KDD*, 2006.