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. Moreover, 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, but 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 Objec 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 - possibly 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 query interface in the style of Query-By-Example, in which both structured and unstructured data (keywords) might be considered.
Figure 1: Architecture of ObjectRunner

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 advantages of the two-phase querying approach.

ObjectRunner uses several novel data structures and algorithms in order to improve scalability and data usability. 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 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 recognizer; 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 \( t = [(t_1, m_1), \ldots, (t_n, m_n)] \) where \( t_1 \) 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 \( t \). For each \( t_i \), \( m_i \) for at least \( m_i \) and at most \( m_i \), 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 addresses 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 \( \tau \) 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 \( t = \{t_i, m_i\}_{i \in T} \) appearing in \( s \), \( \tau \) defines a separator string \( sep \); it denotes that consecutive instances of \( t_i \) will be separated by this string.
- for each tuple type \( t = \{t_1, \ldots, t_k\} \), \( \tau \) defines a total order over the collection of types and a sequence of \( k + 1 \) separator strings \( sep_1, \ldots, sep_{k+1} \); this denotes that the \( k \) instances of the \( k \) types forming \( t \), in the specified order, will be delimited by these separators.

The \( s \) pages 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 \( \tau_i \) 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, the objects that are lost could very likely 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.
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 Tidy [10] 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 (similar to 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 vast 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 Group class, which itself is related to Artist one by a isMemberOf relationship. 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 t1, . . . , tn 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 Si, 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′ ⊆ 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 Si, its output is the extraction template τi 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
approaches extract only the data that the user is interested in, using to their automation degree (for a survey, see [11]). The manual existing works in Web data extraction can be classified according to the demonstration.

On the sample pages of our example, if annotations are taken into account, we can detect that the \(<div>\) tag occurrences denoted div1, div2, and div3 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:

\[
\text{\langle html\rangle}
\text{\langle body\rangle}
\text{\langle div type=\text{\textsc{artist}}\rangle} \quad \text{*} \
\text{\langle div type=\text{\textsc{date}}\rangle} \quad \text{*} \
\text{\langle div type=\text{\textsc{address}}\rangle} \
\text{\quad \langle span\rangle} \quad \text{\text{*}\langle/\text{span}\rangle} \
\text{\quad \langle/\text{span}\rangle} \
\text{\quad \text{*}} \
\text{\langle/\text{div}\rangle}
\text{\text endl/\text{body}\rangle}
\]

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. 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 as well.

For space reasons, other aspects of the system such as ranking and de-duplication were not presented in this paper, but will be discussed during the demonstration.

5. RELATED WORK
The existing works in Web data extraction can be classified according to their automation degree (for a survey, see [11]). The manual approaches extract only the data that the user is interested in, using wrapper programming languages (XWrap [12]) or visual platforms to construct extraction programs (e.g., Lixto [7]). These approaches cannot scale to a large number of sites due to significant manual labeling efforts. The supervised approaches use learning techniques called wrapper induction to learn data extraction rules from a set of manually labeled pages (OLERAl 3], Thresher [9]). 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 pages’ regularity. One important issue is how to distinguish the role of each token, which could be either a piece of data a component in the page encoding template. Some, as a simplifying assumption, consider that every HTML tag is generated by the template (as in DeLa [15], DEPTA [16]). Road-Runner [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 token are those corresponding to frequently occurring equivalence class. Moreover, it has the most general approach, as it can handle optional and alternatives parts of pages. TurboSyncer [5] is an integration system which can incorporate many sources and uses existing extraction results to better calibrate future extractions. For a survey on the management of information extraction systems see [13] and the references therein.

6. REFERENCES