Automatic Extraction of Structured Web Data with Domain Knowledge

Nora Derouiche, Bogdan Cautis, Talel Abdessalem

Télécom ParisTech - CNRS LTCI
Paris, France
firstname.lastname@telecom-paristech.fr

***Abstract*—We present in this paper a novel approach for extracting structured data from the Web, whose goal is to harvest real-world items from template-based HTML pages (the structured Web). It illustrates a two-phase querying of the Web, in which an intentional description of the data that is targeted is first provided, in a flexible and widely applicable manner. The extraction process leverages then both the input description and the source structure. Our approach is domain-independent, in the sense that it applies to any relation, either flat or nested, describing real-world items. Extensive experiments on five different domains and comparison with the main state of the art extraction systems from literature illustrate its flexibility and precision. We advocate via our technique that automatic extraction and integration of complex structured data can be done fast and effectively, when the redundancy of the Web meets knowledge over the to-be-extracted data.**

I. Introduction

We are witnessing in recent years a steady growth of the *structured Web*, documents (Web pages) that are data-centric, presenting structured content, complex objects. Such schematized pages are mostly generated dynamically by means of formatting templates over a database, possibly using user input via forms (in hidden Web pages). In addition, there is also strong recent development of the collaborative Web, representing efforts to build rich repositories of user-generated structured content. Exploiting this significant and rapidly increasing portion of the Web 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.

Extracting data from pages that (i) share a common schema for the information they exhibit, and (ii) share a common template to encode this information, is significantly different from the extraction tasks that apply to unstructured (textual) Web pages. While the former harvest (part of) the exhibited data mainly by relying on "placement" properties w.r.t. the sources' common features, the latter usually work by means of textual patterns and require some initial bootstrapping phase (e.g., positive instances).

A generic methodology for extracting structured data from the Web, followed in most recent works, consists (in order) of (a) the identification of the data rich regions within a page, (b) the extraction of information, and (c) the semantic interpretation of the extracted data. This methodology is well adapted to scenarios where a complete mapping into tables of the data rendered in `HTML` pages is required.

The techniques that apply to schematized Web sources are generally called *wrapper inference* techniques, and have been extensively studied in the literature recently, 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], [2], [3]. They are all illustrations of the aforementioned methodology, 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 then used to populate a relational-style table, a prior without any knowledge over its content. Semantic criteria are taken into account only in subsequent steps, e.g., for column labeling, and this final step can be done either by manual labeling or even by automatic post-processing (a non-trivial problem in its own, [4]).

In practice, this generic methodology suffers from two significant shortcomings, which often limit the usability of the collected data in real-life scenarios:

- only part of the resulting data may be of real interest for a given user or application, hence considerable 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 prone to failure.

These shortcomings seem however avoidable when one has initial knowledge about the to-be-extracted data - in what could be seen as *targeted wrapping and extraction* - and uses this knowledge to guide the extraction process. For this natural and conceptually simple setting, we propose in this paper the `ObjectRunner` system for wrapping structured Web sources.

Our system is attacking the wrapping problem from the angle of users looking for a certain kind of information on the Web. 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 and flexible manner (e.g., provided via a query interface). The interest of having such a description is twofold: it allows to improve the accuracy of the extraction process (as shown in our empirical evaluation), in many cases quite significantly, and it makes this process more efficient and lightweight by avoiding unnecessary computations.

To place the targeted extraction problem - the main focus of this paper - in the more general context of the ObjectRunner system, we give in Figure 1 a high level view of its architecture (mostly self-explanatory). For start, information seekers express what could be called a "phase-one query", in which they are expected to specify by means of an SOD (to be formally defined shortly) what must be obtained from Web pages. It does not of course say *how* this data could be extracted. In particular, the input query (SOD) says what atomic types (i.e., simple entities) are involved in the intentional description and how (e.g., occurrence constraints, nesting). For this, we consider how the intentional description of the targeted data can be formulated in a flexible and widely-applicable manner. We also discuss how such a description can be made effective with little burden on the user, by automatically exploiting rich sources of raw information that are nowadays ubiquitous (for instance, Wikipedia-based ontologies or Web-based textual corpuses) as means to recognize atomic data of interest. Other aspects illustrated in Figure 1, such as source selection and pre-processing of extracted data (e.g., de-duplication), integration and "phase-two" querying of the data, go beyond the scope of this paper. *For the purposes of this paper, we simulate the step of finding relevant sources for a given SOD by "computing with humans" in the Mechanical Turk platform (more details on this in the experiments section).*

To the best of our knowledge, our work is the first to consider the challenges associated with the targeted extraction setting - in which an intentional description of the data is provided before the extraction phase - and to evaluate its advantages in practice, over real-world data. Besides presenting our approach for the extraction of complex objects under the specification of domain knowledge and its empirical evaluation, a goal of this paper is to advocate the potential of this two-phase querying paradigm for the Web. Our extensive experiments indicate 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. They are made even more relevant by the fact we did not manually select the sources that were tested, as these were provided for each domain by independent users (Mechanical Turk workers). In our view, these 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.

The rest of the paper is organized as follows. In the next section we introduce the necessary terminology and some technical background. In Section III we model the problem and give a detailed description of the extraction procedure in terms of composing parts and implementation approaches. We also illustrate how it operates through several examples. The experimental evaluation is presented in Section IV. In Section V we discuss the most relevant related works and we conclude in Section VI.

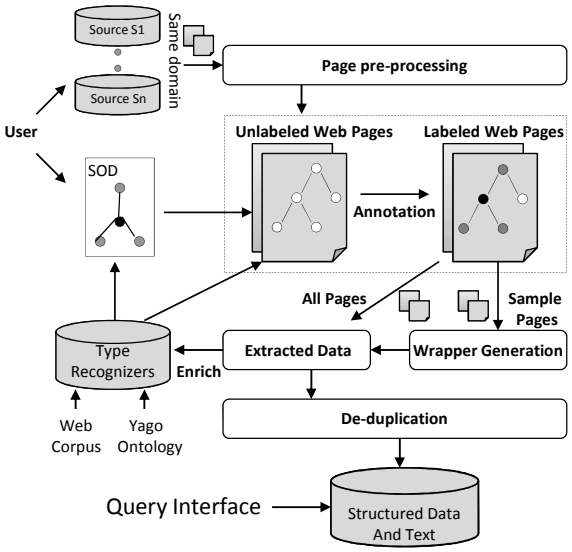


Fig. 1. Architecture of ObjectRunner

II. Preliminaries

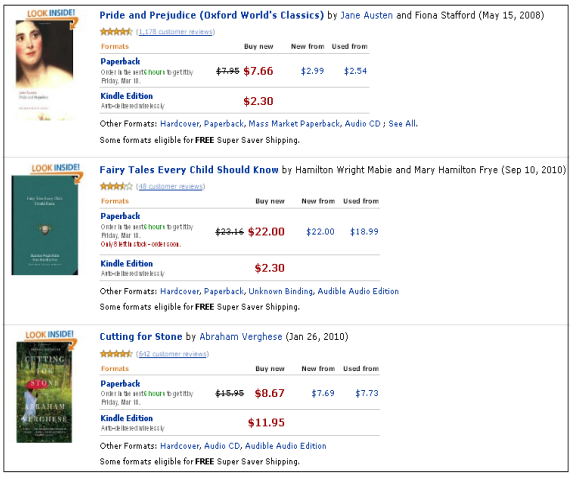
We introduce in this section the necessary terminology and some technical background.

Schematized, data-rich pages are generally built by a script that retrieves information from an underlying databases. For the purpose of extraction, it is useful to differentiate between two types of such pages. *List* pages encode a list of similar records or objects, with both the high-level structure and the individual records being formatted using the common template. An example of list page is given in Figure 2(a), for an amazon.com page displaying details about books. *Singleton* pages focus on a single object (see the page of Figure 2(b)) and, while being structured, in general give a more detailed description of the data. It is quite frequent to have both kinds of pages appearing in the same Web site. The former serves information in a distilled manner (e.g, our example page gives the main description, such as title, author, etc), while the latter complement the list pages by giving more details (e.g., shipping details).

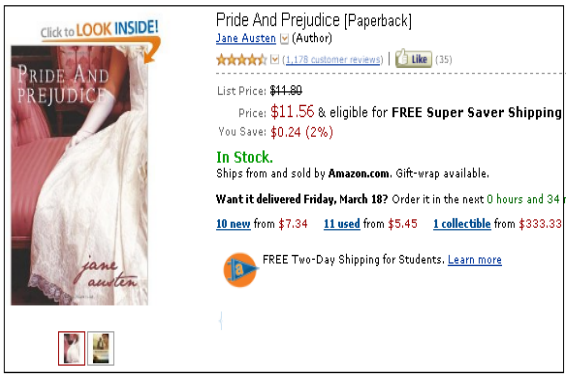
For the purposes of wrapping, although dealing with the same data and appearing in same Web site, the two kinds of pages will be considered as representing distinct sources, given that they rely on different templates and they encode data at different granularities.

As part of the input for the extraction, we suppose that the user has a number of structured Web sources, denoted in the following $\{S_1, \ldots, S_n\}$, where each 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 could be seen as relational (flat) tuples formed by three atomic type values: artist, date, address. We illustrate in Figure 3 three fragments of template-based pages that describe such information.

We continue by defining the typing formalism, by which any user can specify what data should be targeted and extracted from HTML pages. We then describe the extraction problem.



(a) A fragment of a list page with three data records



(b) A segment of a detail page

Fig. 2. Structured pages from amazon.com

A. Types and object description

In short, Structured Object Descriptions allow users to describe *nested relational data with multiplicity constraints* [5], starting from atomic types (not necessarily predefined). More precisely, as building blocks for describing data, we assume a set of *entity (atomic) types*, where each such type represents an *atomic* piece of information, expressed as a string of tokens (words). Each entity type t_i is assumed to have an associated *recognizer* r_i which can be simply viewed as a regular expression or a dictionary of values. For most practical cases, we 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 (called hereafter *isInstanceOf* recognizers). We discuss possible recognizer choices, the ones we experimented with in this paper and possible implementations in the next section.

Based on entity types, *complex types* can be defined in recursive manner. 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, 1 for exactly one. A *tuple type* denotes an unordered collection of set or tuple types. A *disjunction type* denotes a pair of mutually exclusive types.

A *Structured Object Description (SOD)* denotes any complex type. In practice, these could be complemented by additional restrictions in the form of value, textual or disambiguation rules[1]. For brevity, these details are omitted in the model described here and in our experiments.

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, and can be viewed as a finite tree whose internal nodes denote the use of a complex type constructor. 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 entity types would be associated to predefined recognizers (for addresses and dates respectively), since this kind of information has easily recognizable representation patterns, while the last one would have an *isInstanceOf* recognizer. For most practical cases, the three components can be given multiplicity 1.

B. The extraction problem

For a given SOD s and source S_i, a *template* τ with respect to 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 input consisting of an 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,

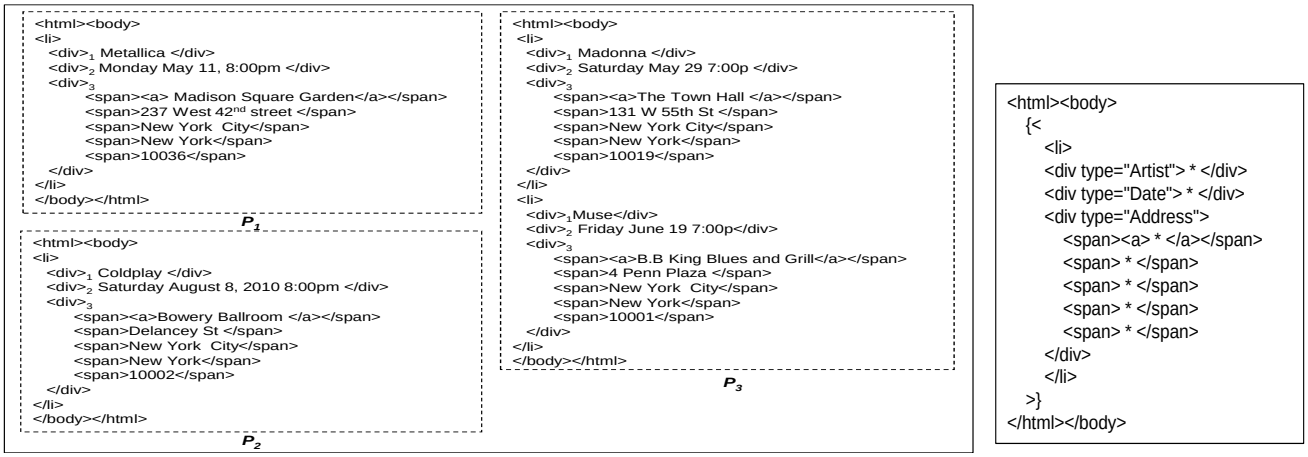
We give in Figure 3(b) the extraction template that would be inferred in our example.

C. Advantages of two-phases querying

As argued in Section I, existing unsupervised approaches have significant drawbacks due to their genericity. We believe that the alternative approach of two-phase querying can often be more suitable in real life scenarios, offering several advantages such as:

Avoiding to mix different types of information. By relying on type recognizers to annotate input pages, we can improve

[1]For instance, these could 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 coordinates



(a) Sample pages

(b) The correct wrapper

Fig. 3. Running example

the extraction process, using semantics besides the structural features of the data.

Extracting only useful data. The description of targeted objects allows us to avoid the extraction of unnecessary data and to preclude any filtering/labelling post-processing.

Stopping early the extraction process. During the process of building the template, if the collection of pages is deemed no longer relevant for the extraction, the wrapper inference can be stopped.

Enabling a self-validating process. Using semantic annotations on the pages' content, we can automatically estimate the quality of the extraction (and of the corresponding wrapper) by checking whether conflicting annotations appear on values of the same attribute. To this end, we rely on an execution loop which can variate certain parameters of the wrapping algorithm.

Avoiding the loss of useful information. Relevant data that should be selected in result objects may be considered "too regular", hence part of the page's template, by techniques that are oblivious to semantics. By consequence, useful data may be missed. For instance, in the running example (Figure 3) the text "New York" appears often, always in the same position in pages, simply because there are many concerts taking place in this city. However, if the text is recognized as denoting an address, a component of concert objects, it will interpreted as information that might be extracted.

Using applicable semantic annotations to discover new (potential) annotations. Unavoidably, the use of dictionaries (gazetteers) to annotate data yields incomplete annotations in practice. But we can use current annotations to discover others based on the extracted data, e.g., enriching in this way the dictionaries.

III. The extraction approach

We describe in this section the main aspects of our extraction approach, in terms of composing parts and implementation choices. 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 fast-performing, yet are flexible enough and have high precision and satisfactory recall, should be favored in most aspects of the system. Though in practice this means that some sources may be handled in unsatisfactory manner, the objects that are lost could very likely be found in another source as well (even within the same Web site) 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.

We first discuss some pre-processing steps. Often, there are many segments in Web pages that do not encode useful information, such as headers, scripts, styles, comments, images, hidden tags, white spaces, tag properties, empty tags, etc. This content can make the later processing slower and sometimes might even affect the end results. Therefore, a pre-processing cleaning step is usually performed on the `HTML` before the extraction process can start. First, beyond page cleaning, we apply to 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 exploit the visual (rendering) information about the page. We rely on an algorithm that performs page segmentation (in the style of VIPS [6] and ViNTs [7]). In VIPS, each page is represented as a "tree structure" of blocks. These blocks are delimited based on : (i) the DOM tree of the page, and (ii) the separators between them, using the horizontal and vertical lines of the web pages. For all our sources, using a rendering engine for the `HTML` pages, we applied the straightforward heuristic of selecting as the best candidate segment the one described by the largest and most central rectangle in the page. As block sizes and even the block structure may vary from one page to another, across all the pages of a given source, we identified the best candidate block by its tag name, its path in the DOM tree and its attribute names and values.

Second, since HTML documents are often not well-formed, we use the open source software JTidy [8] to transform them to XML documents. For instance, the simplified pages in our example were obtained after such pre-processing steps from the site http://upcoming.yahoo.com/.

A. Type recognizers

Importantly, type recognizers are never assumed to be entirely precise nor complete by our algorithm. 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 is provided, without explicit means to recognize instances thereof. This could be the case for the *Artist* entity type. When such a type is required by the user, an important functionality of ObjectRunner is that it should seamlessly construct on the fly a dictionary-based recognizer for it. For this, we experimented for this paper's results two complementary ways, which also follow the guideline of fast-performing, conceptually-simple best-effort approaches.

The first alternative is the one of querying a knowledge base. In our experiments, we used the YAGO ontology [9], 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.

A second alternative is to look for instances of a given type (specified by its name) directly on the Web. To support this feature, we chose to apply Hearst patterns [10] on a corpus of Web pages that is pre-processed for this purpose. These are simple parameterized, textual, patterns like *Artist such as X*, or *X is an Artist*, by which one wants to find the values for the *X* parameter in the text.

Regardless of how they are obtained, gazetteer instances should be described by confidence values w.r.t. the type their are associated to. While YAGO comes with its own confidence values, for the second alternative of populating a gazetteer, given a candidate instance i and type t, a confidence score for each pair (i,t) is computed in ObjectRunner using the Str-ICNorm-Thresh metric of [11]:

$$score(i,t) = \frac{\sum_{p \in P} count(i,t,p)}{max(count(i), count_{25}) \times count(t)} \qquad (1)$$

In this measure, $count(i,t,p)$ is the number of hits for the pair (i,t) in the corpus by the pattern p, $count(i)$ is the number of hits for the term i in the corpus, and $count_{25}$ is the hit count that occurs at the 25th percentile. Intuitively, if this pair is extracted many times in the corpus, this redundancy gives more confidence that the pair is correct.

Such rich sources of information (Wikipedia-based ontologies or large Web-based textual datasets such as ClueWeb[2]) are nowadays ubiquitous and can be easily employed as means to recognize atomic data, without the need to declare beforehand predefined types of interest.

B. Annotation and page sample selection

No a prior 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 (approximately 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).

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 (equation 2)
4: for all entity types t in s do
5:     look for matches of t in S and annotate
6:     compute score of each page
7:     for S' ⊑ S top annotated pages, make S := S'
8: end for
9: return sample as most annotated k pages in S
```

Annotation. Given multiple dictionaries, each with the associated confidence scores (this could be seen as a large disjunctive keyword query), the top pages with respect to annotations are selected and used as training sample to construct the extraction template. The annotation is done by assigning an attribute to the DOM node containing the text that matched the given type. Multiple annotations may be assigned to a given node.

The result will be a type-annotated DOM tree. For instance, in page p_1 of our example, the first tag contains an artist name, so it will be annotated accordingly, as in Metallica . Annotations will also be propagated upwards in the DOM tree to ancestors as long as these nodes have only one child (i.e., on a linear path) or all children have the same annotation.

Page sample selection. In this step, the top annotated pages are selected for wrapper inference. We first associate to each *isInstanceOf* type t a selectivity estimate, computed

[2]http://www.lemurproject.org/clueweb09.php.

as follows:

$$score(t) = \sum_{i \in t} score(i,t)/tf(i), \quad (2)$$

where i denotes each instance in the dictionary associated with type t, described by its confidence score and a term frequency $tf(i)$ (from either the Web corpus or the ontology). We then apply types in the annotation rounds in decreasing order of their selectivity estimate.

After each annotation round, we continue the matching process only on the "richest" pages. For that, we order the pages by their minimum score with respect to the types that were already processed, as $min(score(page/t_1), \ldots, score(page/t_n))$ and we select the pages having the highest score. The score per page is computed as follows:

$$score(page/t_j) = \sum_{i' \in t_j} score(i,t_j)/tf(i) \quad (3)$$

where i' denotes each instance of t_j in the page.

During this loop, we strive to minimize the number of pages to be annotated at the next round. Moreover, the source could be discarded if unsatisfactory annotation levels are obtained. Once the top annotated pages are selected over all *isInstanceOf* types, the predefined and regular expression types are processed.

C. Wrapper generation

This is the core component of our algorithm for targeted extraction. For each source S_i, it outputs an 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].

A template is inferred from a sample of source pages based on *occurrence vectors* for page tokens (words or `HTML` tags) and *equivalence classes* defined by them. An equivalence class denotes a set of tokens having the same frequency of occurrences in each input page and a role that is deemed *unique* among tokens. For example, the token `<div>` has three occurrences in the two first pages and six occurrences in the third page of our running example, and this would correspond (initially) to the following vector of occurrences: $< 3,3,6 >$. Such descriptions can be seen as equivalence classes for tokens, and equivalence classes determine the structure that is inferred from Web pages. Hence determining the roles and distinguishing between different roles for tokens becomes crucial in the inference of the implicit schema, and in [1] this depended on two criteria: the position in the DOM tree of the page and the position with respect to each equivalence class that was found at the previous iteration. Consecutive iterations refine the equivalence classes until a fix-point is reached, while at each step the invalid classes are discarded (following the guideline that information, i.e. classes, should be properly ordered or nested).

How roles and equivalence classes are computed distinguishes our approach from [1]. First, we use annotations as an additional criterion for distinguishing token roles. 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 can occur, as the annotations alone do not guarantee success), and (ii) accept approximate equivalence classes outside the ones that might represent to-be-extracted instances.

As annotations are used to further distinguish token roles, 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:      if abort conditions are verified(*) then
7:        stop process
8:      end if
9:      differentiate roles using EQs + non-conflicting annotations
10:   until fixpoint
11:   differentiate roles using EQs + conflicting annotations
12: until fixpoint
```

(*) Details given in Section III-E.

First, roles of tokens are determined using the HTML format of the page (line 1): tokens having the same value and the same path in the DOM will have the same role. Then, more refined roles of tokens are assigned in the loop, based on appearance positions in equivalence classes (line 3-10). During this loop, all invalid equivalence classes are discarded and the process can be stopped if certain conditions are not verified (Section III-E).

Using the annotations is an obvious strategy in our context but, since these annotations are not complete and might even be conflicting over the set of pages, this strategy needs to be applied cautiously. We can distinguish two types of annotations:

- **Non-conflicting annotations.** A token - identified by its DOM position and its coordinates with respect to the existing equivalence classes - has non-conflicting annotations if each of its occurrences have the same (unique) type annotation or no annotation at all.
- **Conflicting annotations.** A token has conflicting annotations if different type annotations have been assigned to its occurrences.

In the first case, tokens without conflicting annotations are treated in the loop along with the other criteria (line 9). Once all equivalences classes are computed in this way, we perform one additional iteration to find new occurrence vectors and equivalence classes. The entire process is then repeated until a fix-point is reached (line 2-12).

Going back to our running 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 of the input SOD. This would not be possible if only the positions in the HTML tree and in equivalence classes were taken into account, as the three occurrences would have the same role. This would lead for instance to the following (incorrect for our purposes) extraction result from page P_1: (1) Metallica Monday May 11, 8:00pm, (2) Madison Square Garden, (3) 237 West 42nd street, and (4) 10036.

For further illustration, there are other opportunities where the annotations can improve the result of the extraction. As a simple example, consider the Amazon.com fragment of the list page in Figure 2(a), which shows three book records. A book can have one or several authors, and they are represented differently in HTML code. We detail below the corresponding HTML for the author part of these books b_1, b_2 and b_3:

b_1: by <a>Jane Austen and Fiona Stafford

b_2: by Hamilton Wright Mabie, Mary Hamilton Frey

b_3: by <a>Abraham Verghese

Taking into account only the positions of tokens in equivalence classes can result in extracting author names as values of distinct fields of the template. But with the annotation inherited by the tag we can determine that this field represents author names and extract its content accordingly.

Finally, while annotations allow us to differentiate the roles of tokens that are in the same position, for a given position, the number of consecutive occurrences of tokens can vary from one page to another. In our running example, the token had the same number of occurrences in each record, but this is not always the case. When this happens, we chose to settle on the minimal number of consecutive occurrences across pages, and differentiate roles within this scope. Once this is chosen, we deal with incomplete annotations by generalizing the most frequent one if beyond a given threshold (0.7 in our experiments).

Enriching the dictionaries. The discovery of new instances during the extraction phase from the Web pages also enables us to enrich our dictionaries. In this regard, we associate confidence score before adding them in the dictionaries based on confidence score from the wrapper generation step, extracted instances (I) and existing instances (D):

$$score(c) = f(wrapper\_score(c), \frac{\sum_{D \cap I} score(i,c)}{count(I)}) \quad (4)$$

This formula gives more weight either to instances obtained by a good wrapper (in short, one built with no or very few conflicting annotations) or to those which have a significant overlap with the set of existing values in dictionaries. Moreover, we can update the scores on existing dictionary values after each source is processed.

D. The output template

The input of the template construction step is a hierarchy of valid equivalence classes $\{EQ_1, \ldots, EQ_n\}$ computed by Algorithm 2. Recall that a valid equivalence class is ordered and any two classes are either nested or non-overlapping. Also recall that each class is described by separators: for instance, a tuple type of size k will have a sequence of $k+1$ separator strings. Then, the corresponding template τ can be represented as a similar tree structure, which can be obtained from the hierarchy of classes by replacing each class by its separators and the type annotations on them. We call this the *annotated template tree*.

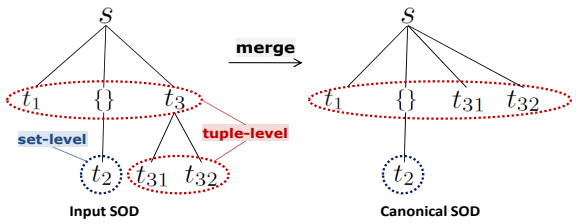


Fig. 4. Bottom-up matching of the SOD.

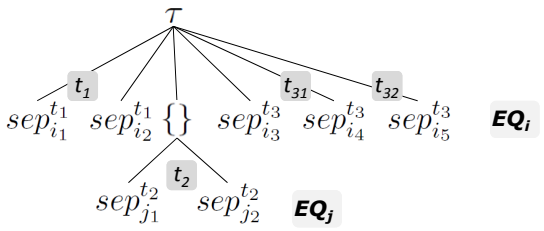


Fig. 5. Annotated template tree.

As an example, consider the template tree in Figure 5, which should match the input SOD s shown in the left part of Figure 4, which is composed of (i) two tuple types $\{t_1, \{\}, t_3\}$ and $\{t_{31}, t_{32}\}$, and (ii) a set type $\{t_2\}$ with arbitrary multiplicity. The root of the template tree covers the entire page, the first level of the tree corresponds to an equivalence class EQ_i described by separator strings and annotations for atomic types t_1, t_{31}, and t_{32}, and the subtree from the second level corresponds to a class EQ_j and is annotated by the atomic type t_2. Each level of the template tree represents a sequence of ordered equivalence classes, and nesting is introduced by the set types.

Given the SOD, we want to decide at this point what should be extracted from the pages. So the task at hand is to identify in the template tree the region(s) (subtrees) that match the targeted SOD. Only these will be extracted. For that, for presentation simplification, we first assume a preliminary step which transforms the input SOD into a canonical form, which essentially groups in the same tuple type all the atomic types that have the same multiplicity. Recall that an SOD can be formed by unordered tuple types and set types and that leaves correspond to atomic types.

In short, to put an SOD in its canonical form, any tuple node will receive as direct children all the atomic-type nodes that are reachable from it only via tuple nodes (no set nodes). We illustrate in the rest of Figure 4 this transformation towards the canonical form: $\{t_{31}, t_{32}\}$ will be combined with $\{t_1, \{\}, t_3\}$ and replaced by a new tuple $\{t_1, \{\}, t_{31}, t_{32}\}$, as shown in the canonical SOD in Figure 4.

Matching step. We then do the matching of the canonical SOD with the template tree bottom-up, by a dynamic programming approach which starting from the leaf classes bearing type annotations, tries to identify a sub-hierarchy that matches the entire SOD.

We start with the SOD's nodes at depth 1 (parents of leaves), which can correspond to either a *tuple-level* or a *set-level*. In the simplest case, if the SOD has a *flat structure* then its tree will have one tuple-level containing all the atomic types as leaves. These atomic types of the SOD should match separators that (i) belong to the same equivalence class, and (ii) have annotations for these types. In the case of a nested structure, matchings of lower levels are combined.

Over our example, at the end, the separators around the atomic types $\{t_1, t_{31}, t_{32}\}$ should appear in the same equivalence class and the ones for the set type $\{t_2\}$ should appear in separate class, nested in the former class.

E. Stopping early the wrapper generation

Having an intentional description of the data to be extracted allows us also to stop the wrapper generation process, when one can understand that it will not yield satisfactory results. We describe in this section how this is done in our system. We can distinguished two phases where the generation process could be stopped.

During the annotation phase. The source could be discarded if unsatisfactory annotation levels are obtained. We have seen previously that for a given SOD, we select a sample of pages that has a satisfactory rate of annotations. Our condition for stopping relies on a threshold and works at the finer granularity of visual blocks in pages. Since different blocks usually represent different contexts in a page, it is safe to distinguish annotations appearing in distinct blocks, even if they represent the same atomic type. For each block, we check if the following condition holds:

$$\sum_{i=1}^{k} \frac{no.\ of\ annotations\ in\ block}{k} > \alpha$$

where k is the size of the sample and α is a threshold (50% in our experiments).

After each annotation round, if we obtain at least one block that satisfies the given condition, we continue the matching process for the remaining types. Otherwise the process is stopped.

During the wrapper generation phase. Recall that the wrapper inference is mainly based on the discovery of equivalence classes and their matching with the specified SOD. We have seen in the previous section that the process for wrapper generation is iterative and it alternates finding EQs, handling invalid EQs and differentiating token roles. At each iteration, the process can be stopped if we can conclude that the current equivalence classes cannot lead to a class hierarchy on which the SOD would match (see Algorithm 2).

For that, after the step of constructing equivalence classes and handling the invalid ones, we consider the template tree corresponding the current (not necessarily final) class hierarchy. We found that, on this tree, the following property should always be verified in a potentially successful wrapper generation process: *there must exist at least one partial matching of the SOD into the current template tree*. In a partial matching, part of the SOD matches (as described in the matching step for template construction, Section III-D), while for each of the missing parts, for each atomic type in them, there is still some untreated token annotated by that type (this token might serve as image for a complete match later on). We test this condition in incremental manner, keeping track of the current partial matchings and their possible completions to a full matching.

IV. Experimental evaluation

We present in this section the experimental results for our targeted extraction algorithm. We focus mainly on the precision of extraction, comparing our results with those of the two publicly available prototypes for wrapping structured Web sources, `ExAlg` and `RoadRunner`. Regarding time performance, we mention that once the necessary recognizers are in place, the wrapping time of our algorithm ranged from 4 to 9 seconds. Once the wrapper is constructed, the time required to extract the data was negligible for all the tested sources. The experiments were conducted on a workstation equipped with a 2.8 GHz CPU and 3GB of RAM.

A. Datasets

The experiments were performed over five different domains: `concerts`, `albums`, `books`, `publications` and `cars`.

To avoid bias towards certain source characteristics, we strived to use in our experiments a selection of Web sources that is both *relevant* and as *objective* as possible. This is why, as a mean to simulate the step of automatic retrieval of most relevant sources for each domain, we used the Amazon's *Mechanical Turk* platform. For all domains, except the one of publications, we asked ten workers to give us a ranked list of ten *browsable* Web sources (as opposed to sources behind query forms) for records from that domain. We then took the top ten sources appearing in the lists provided by the workers. An exception was made for concerts, which had fewer responses and gave us a top five of sources[3]. For scientific publications, for which we considered that the Mechanical Turk approach may be less adequate, we used instead the complete list of sources cited in the experiments of [12]. Finally, to the sources chosen in this way, we also added the ones used in the TurboSyncer [13] wrapping system, on books, albums and cars respectively.[4]

Then, the datasets for which we report the extraction results are those which passed the rendering-based simplification step discussed in the beginning of Section III. Without going into further detail on this preliminary step, we mention that in more

[3]The detailed answers from Mechanical Turk are available on our project page http://perso.telecom-paristech.fr/~derouich/datasets/

[4]We were unable to obtain the prototype implementation or the detailed results per source from the authors of [13].

TABLE I
EXTRACTION RESULTS

	Domains	Sites	Attributes				Objects			
			Optional	A_c	A_p	A_i	N_o	O_c	O_p	O_i
1.	Concerts	zvents (detail)	yes	4/4	0/4	0/4	50	50	0	0
2.		zvents (list)	yes	4/4	0/4	0/4	150	150	0	0
3.		upcoming.yahoo (detail)	yes	4/4	0/4	0/4	50	50	0	0
4.		upcoming.yahoo (list)	yes	3/4	0/4	1/4	250	0	0	250
5.		eventful (datail)	yes	1/4	2/4	1/4	50	0	0	50
6.		eventful (list)	no	3/4	0/4	0/4	500	500	0	0
7.		eventorb (detail)	yes	4/4	0/4	0/4	50	50	0	0
8.		eventorb (list)	yes	4/4	0/4	0/4	289	289	0	0
9.		bandsintown (detail)	yes	4/4	0/4	0/4	50	50	0	0
10.	Albums	amazon	yes	4/4	0/4	0/4	600	600	0	0
11.		101cd	no	1/4	2/4	0/4	1000	0	1000	0
12.		towerrecords	yes	4/4	0/4	0/4	1250	1250	0	0
13.		walmart	yes	3/4	1/4	0/4	2300	0	2300	0
14.		cdunivers	yes	4/4	0/4	0/4	1700	1700	0	0
15.		hmv	yes	4/4	0/4	0/4	600	600	0	0
16.		play	no	3/4	0/4	0/4	1000	1000	0	0
17.		sanity	yes	4/4	0/4	0/4	2000	2000	0	0
18.		secondspin	yes	4/4	0/4	0/4	2500	2500	0	0
19.		emusic (discarded)								
20.	Books	amazon	yes	4/4	0/4	0/4	600	600	0	0
21.		bn	yes	4/4	0/4	0/4	500	500	0	0
22.		buy	no	3/4	0/4	0/4	1300	1300	0	0
23.		abebooks	no	3/4	0/4	0/4	500	500	0	0
24.		walmart	yes	3/4	0/4	1/4	2300	0	0	2300
25.		abc	yes	4/4	0/4	0/4	651	651	0	0
26.		bookdepository	yes	4/4	0/4	0/4	1000	1000	0	0
27.		booksamillion	yes	4/4	0/4	0/4	1000	1000	0	0
28.		bookstore	no	2/4	0/4	1/4	730	0	0	730
29.		powells	no	3/3	0/3	0/3	1000	1000	0	0
30.	Publications	acm		3/3	0/3	0/3	1000	1000	0	0
31.		dblp		3/3	0/3	0/3	500	500	0	0
32.		cambridge		3/3	0/3	0/3	230	230	0	0
33.		citebase		3/3	0/3	0/3	500	500	0	0
34.		citeseer		1/3	2/3	0/3	500	0	500	0
35.		DivaPortal		3/3	0/3	0/3	500	500	0	0
36.		GoogleScholar		1/3	0/3	2/3	500	0	0	500
37.		elseivier		3/3	0/3	0/3	983	983	0	0
38.		IngentaConnect		2/3	0/3	1/3	500	0	0	500
39.		IowaState		0/3	0/3	3/3	481	0	0	481
40.	Cars	amazoncars		2/2	0/2	0/2	54	54	0	0
41.		automotive		0/2	2/2	0/2	750	0	750	0
42.		cars		2/2	0/2	0/2	500	500	0	0
43.		carmax		2/2	0/2	0/2	500	500	0	0
44.		autonation		2/2	0/2	0/2	500	500	0	0
45.		carsshop		2/2	0/2	0/2	500	500	0	0
46.		carsdirect		0/2	2/2	0/2	1500	0	1500	0
47.		usedcars		2/2	0/2	0/2	1250	1250	0	0
48.		autoweb		2/2	0/2	0/2	250	250	0	0
49.		autotrader		2/2	0/2	0/2	393	393	0	0

than 80% of the cases our heuristic reduces away the non-significant segments of the pages.

Dictionary completeness. In practice, we often cannot hope to have a complete dictionary for *isInstanceOf* types (e.g., for book or album titles). Regarding this aspect, the main goal in our experiments was to show that, when dictionaries ensure a reasonable covering of the pages content, the extraction algorithm performs well. To this end, when necessary, we completed each dictionary in order to have at least 20% of the instances from a given sources (we also report in the Appendix A of the extended version [14] the experimental results for 10% dictionary coverage).

For each source, we randomly selected roughly 50 pages. As the sources we selected propose in general list pages on the surface, we focused mainly on such pages. In order to illustrate that our techniques performs well on both types of pages, for the concert domain, we collected also singleton pages from each source. The pages selected for each source are produced by the same template, but we stress that one of the advantages of our approach is that we can detect when a source is not relevant or not structured enough and we can discard it without producing low-quality results.

For each domain, the respective sources were matched with SODs as follows:

1) **Concerts.** A concert object is described by a two-level tree structure. At the first level, it is composed of (i) three entity (leaf) types, *artist* and *date*, and (ii) a tuple type denoted *location*. The latter is in turn composed of two entity types, *address* and *theater name*. The address is considered optional.
2) **Albums.** An album object is described as a tuple composed of four entity types: *title*, *artist*, *price* and *date*. The date is considered optional.
3) **Books.** A book object is described by a two-level tree structure. At the first level, we have a tuple composed of three entity types: *title*, *price*, *date*, and one set type, *authors*. The latter denotes a set of instances of the entity type *author*. The date is considered optional.
4) **Publications.** A publication object is described by a two-level tree structure. At the first level, we have a tuple composed of two entity types: *title*, *date*, and one set type, *authors*. The latter denotes a set of instances of the entity type *author*. The date is considered optional.
5) **Cars.** A car object is described by a tuple type composed of two entity types: *car brand* and *price*.

B. Results

The extraction results are summarized in Table I. We report there on the precision of the template construction step - i.e., how many components (entity types) of the SOD were correctly identified in the pages' structure - based on a sample of the top 20 annotated pages. For each source we checked whether the *optional* attribute was present or not in the source (we mark this in the table as well). We then quantify precision using the golden standard test[5]. We classify results as follows:

Correct attributes (A_c) and objects (O_c). *An attribute* is classified as correct if the extracted values for it are correct. *An object* is classified as correct if all the extracted attributes are corrects.

Partially correct attributes (A_p) and objects (O_p). *An attribute* is classified as partially correct if: (i) the values for two or several attributes are extracted together (as instances of the same field in the template) and they are displayed in this manner in pages (for example, a book title and the author name may be appear in text as an atomic piece of information), or (ii) values corresponding to the same entity type of the SOD are extracted as instances of separate fields (this can occur in the case of list pages). *An object* is classified as partially correct if the extracted attributes are only corrects or partially correct.

Incorrect attributes A_i and objects O_i. An attribute is classified as incorrect if the extracted values are incorrect, i.e., they represent a mix of values corresponding to distinct attributes of the implicit schema. *An object* is classified incorrect if there exists at least one incorrect extracted attribute.

For both attributes and objects, we show in Table I the proportion of correct, partially-correct or incorrect results.

[5]We extracted manually objects from each page for comparison.

We use two precision measures to evaluate the quality of extraction: (i) the precision for correctness (P_c), and (ii) the precision for partially-correctness (P_p), defined as follows:

$$P_c = \frac{O_c}{N_o} \quad and \quad P_p = \frac{O_c + O_p}{N_o},$$

where O_c denotes the total number of correctly extracted objects, O_p denotes the number of partially correct extracted objects, and N_o is the total number of objects in the source. (Note that, in our setting, the recall is equal to the precision for correctness, since the number of existing objects equals the number of extracted objects - correct, partially-correct and incorrect.)

Automatic variation of parameters. Given the SOD and the annotations on the pages' content, we were able to automatically estimate the quality of the extraction, by considering the conflicting annotations appearing on values of the same attribute. When necessary, we variate the parameters of the wrapping algorithm and re-execute it. For instance, by variating the *support* (minimal number of pages in which a token should appear, between 3 and 5 pages in our experiments), we managed to improve significantly the precision on publication sources (further details can be found in the Appendix B of the extended version [14]).

1) Sample selection vs. random selection: We analyzed the difference in extraction precision depending on how is selected the sample of pages on which the wrapper is built. For that, we compared the results obtained using the selection approach described previously with the ones of a baseline approach that selects the sample in random manner. The results for the two approaches are summarized in Table II. They support the fact that a careful selection of the sample, depending on the target SOD, can significantly improve the end results.

TABLE II
PRECISION DEPENDING ON HOW THE SAMPLE OF PAGES IS SELECTED: RANDOM VS. SOD-BASED. (%)

	Sample selection		Random selection	
Domains	P_c	P_p	P_c	P_p
Concerts	86.10	86.10	61.78	61.78
Albums	74.52	100	69.88	95
Books	68.37	68.37	56.36	62
Publications	65.21	74	65.21	65.21
Cars	75.79	100	75.79	100

2) Comparison with state-of-the-art approaches: We compared the results of `ObjectRunner` (OR) with two of the most cited and closely related works, namely `ExAlg` [6] [1] (EA) and `RoadRunner` [7] [2] (RR). (Unfortunately, a prototype of another state-of-the-art approach, DEPTA [3], is no longer available for comparison). We report in Table III and Figure 6 the robustness of our extraction approach, in terms of precision, compared with the ones of the two existing algorithms.

[6]The prototype used here was obtained from its authors.

[7]A prototype implementation is publicly available at http://www.dia.uniroma3.it/db/roadRunner.

In Table III we report on both precision values for each domain and source. One can note that ObjectRunner outperforms RoadRunner by a significant margin (roughly about 60%) in all five domains. We found that RoadRunner fails mostly on list pages, where most extracted objects are partially correct (listed values corresponding to the same attributed of the SOD are extracted separately). Surprisingly, RoadRunner fails to handle list pages that are "too regular", when the number of records in the input pages is constant (this is the case in our sources for books and publications, about 50% partially correct). We observe that, overall, ExAlg outperforms RoadRunner, but for many of the sources the former extracts several attributes together. The precision results on the two state-of-the-art system may seem surprisingly low, especially compared with the ones reported by their authors. But one should keep in mind here that this occurs in the context of targeted extraction, where only a (small) fraction of the information displayed in pages is of interest and is accounted for in the precision results.

TABLE III
PERFORMANCE RESULTS (%)

	OR		EA		RR	
Domains	P_c	P_p	P_c	P_p	P_c	P_p
Concerts	86.10	86.10	45.17	45.17	6.95	72
Albums	74.52	100	69.88	95	17.37	82
Books	68.37	68.37	50.10	62	0	50,10
Publications	65.21	74	34.83	56	0	52.39
Cars	75.79	100	75.79	100	15.28	72

For a clearer view, in Figure 6, we provide two facets for extraction quality: (i) the rate of correct, partially correct and incorrect objects that were extracted (depending on the accuracy of the template that was inferred for each source), and (ii) the fraction of sources that were incompletely handled (i.e., with partially correct or incorrect attributes). By the latter, we compare the three algorithms by their ability to handle a source correctly. We observe that in all the three domains OR outperforms EA and RR. Indeed, there are roughly 20% incompletely handled sources for three of the domains (concerts, albums and books), 40% for publications and 10% for cars.

These results are rather encouraging since, in our view, it is far more important to be able to handle correctly at least some sources from a given domain than to handle most or all sources in fair but incomplete (partially) manner. For example, as Web data tends to be very redundant, the concerts one can find in the yellowpages.com site (a source that was initially candidate for our tests) are precisely the ones from zvents.com.

V. RELATED WORK

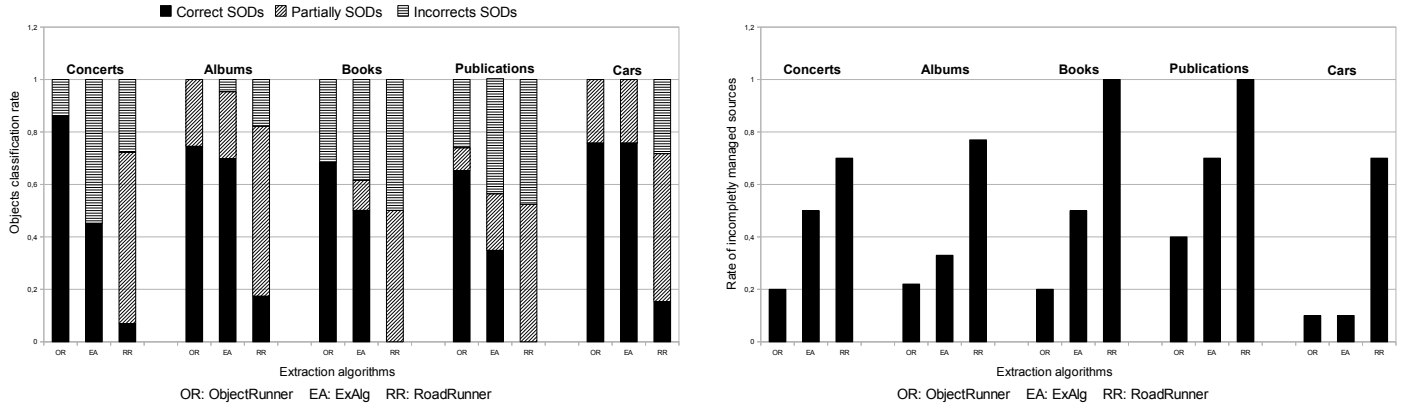
The existing works in data extraction from structured Web pages can be classified according to their automation degree: manual, supervised, semi-supervised and unsupervised (for a survey, see [15]). The manual approaches extract only the data that the user marks explicitly, using either wrapper programming languages, such as the ones proposed in [16], or visual platforms to construct extraction programs, like Lixto [17]. Supervised approaches use learning techniques, called wrapper induction, to learn extraction rules from a set of manually labeled pages (as in WIEN [18], Stalker [19]). A significant drawback of these approaches is that they cannot scale to a large number of sites due to significant manual labeling efforts. Semi-supervised (e.g., OLERA [20], Thresher [21]) arrive to reduce human intervention by acquiring a rough example from the user. Some semi-supervised approaches (such as IEPAD [22]) do not require labeled pages, but find extraction patterns according to extraction rules chosen by the user.

Unsupervised approaches (automatic extraction systems) 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 a component in the encoding template. Some, as a simplifying assumption, consider that every HTML tag is generated by the template (as in DeLa [23], DEPTA [3]), which is often not the case in practice. RoadRunner [2] and G-STM [24], which use an approach based on grammar inference and schema matching also assume that every HTML tag is generated by the template, but other string tokens could be considered as part of the template as well. However, G-STM [24] integrates more robust method to detect lists with ability to handle nested lists. 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 alternative parts of pages. TurboSyncer [13] is an integration system which can incorporate many sources and uses existing extraction results to better calibrate future extractions. Other works explore the mutual benefit of annotation and extraction, learning wrappers based on labeled pages [25], [12] or domain knowledge for query result records [26].

The key advantage of wrapper induction techniques is that they extract only the data that the user is interested in. Due to manual labeling, the matching problem is significantly simpler. The advantage of automatic extraction is that it is applicable at large scale, the tradeoff being that it may extract a large amount of unwanted data. Our approach aims to obtain the best of both works, by exploiting both the structure and user-provided semantics in the automatic wrapper generation process.

Most research works in information extraction from unstructured source focus on extracting semantic relations between entities for a set of patterns of interest. This is usually done by predefined relation types (as in DIPRE [27], KnowItAll [28]), or by discovering relations automatically (TextRunner [29]). Other systems in this area, like Yago [9] and DBpedia [30], extract relation by rule-based harvesting of facts from semi-structured sources such as Wikipedia.

A preliminary version of the ObjectRunner system, focused on aspects such as the specification of SODs, the dictionary build-up and the interrogation interface, was demonstrated



(a) Objects classification

(b) Incompletely managed sources

Fig. 6. `ObjectRunner` comparison

recently in [31].

VI. Conclusion

This paper proposes an alternative approach to automatic information extraction and integration from structured Web pages. To the best of our knowledge, it is the first to advocate and evaluate the advantages of a two-phase querying of the Web, in which an intentional description of the target data is provided before the extraction phase. More precisely, the user specifies a Structured Object Description, which models the objects that should be harvested from `HTML` pages. This process is domain-independent, in the sense that it applies to any relation, either flat or nested, describing real-world items. Also, it does not require any manual labeling or training examples. The interest of having a specified extraction target is twofold: (i) the quality of the extracted data can be improved, and (ii) unnecessary processing is avoided.

We validate through extensive experiments the quality of extraction results, by comparison with two of the most referenced systems for automatic wrapper inference. By leveraging both the input description (for five different domains) and the source structure, our system harvests more real-world items, with fewer errors. Our results are rendered even more relevant by the fact we did not manually select the sources that were tested. These were provided to us by independent users (Mechanical Turk workers), as the most relevant ones for each domain.

Besides the extraction tasks, there are other exciting research problems we are currently investigating in the `ObjectRunner` system. We are currently studying techniques for discovering, processing and indexing structured Web sources (they were simulated by Mechanical Turk tasks for the purposes of this paper). Also, given an input SOD, we would like to be able to automatically select the most relevant and data rich sources. We are also considering the possibility of specifying atomic types by giving only some (few) instances. These will then be used by the system to interact with YAGO and to find the more appropriate concepts and instances (in the style of Google sets).

References

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

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

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

[4] G. Limaye, S. Sarawagi, and S. Chakrabarti, "Annotating and searching web tables using entities, types and relationships," *PVLDB*, vol. 3, no. 1, 2010.

[5] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

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

[7] H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu, "Fully automatic wrapper generation for search engines," ser. WWW, 2005.

[8] JTidy, http://jtidy.sourceforge.net.

[9] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: a core of semantic knowledge," in *WWW*, 2007.

[10] M. A. Hearst, "Automatic acquisition of hyponyms from large text corpora," in *COLING*, 1992.

[11] L. McDowell and M. J. Cafarella, "Ontology-driven, unsupervised instance population," *J. Web Sem.*, 2008.

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

[14] http://biblio.telecom-paristech.fr/cgi-bin/download.cgi?id=11912.

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

[16] S. Soderland, "Learning information extraction rules for semi-structured and free text," *Machine Learning*, 1999.

[17] 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.

[18] N. Kushmerick, D. S. Weld, and R. B. Doorenbos, "Wrapper induction for information extraction," in *IJCAI*, 1997.

[19] I. Muslea, S. Minton, and C. A. Knoblock, "A hierarchical approach to wrapper induction," in *Agents*, 1999.

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

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

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

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

[24] N. Jindal and B. Liu, "A generalized tree matching algorithm considering nested lists for web data extraction," in *SDM*, 2010.

[25] 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.

[26] W. Su, J. Wang, and F. H. Lochovsky, "Ode: Ontology-assisted data extraction," *ACM Trans. Database Syst.*, 2009.

[27] S. Brin, "Extracting patterns and relations from the world wide web," in *WebDB*, 1998.

[28] M. J. Cafarella, D. Downey, S. Soderland, and O. Etzioni, "KnowItNow: fast, scalable information extraction from the web," in *HLT Conference*, 2005.

[29] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni, "Open information extraction from the web," in *IJCAI*, 2007.

[30] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives, "DBpedia: A nucleus for a web of open data," in *ISWC/ASWC*, 2007.

[31] T. Abdessalem, B. Cautis, and N. Derouiche, "ObjectRunner: Lightweight, targeted extraction and querying of structured web data," *PVLDB*, vol. 3, no. 2, 2010.