Lightweight, Targeted Extraction of Structured Web Data

Talel Abdessalem Bogdan Cautis Nora Derouiche

Télécom ParisTech - CNRS LTCI, Paris, France

firstname.lastname@telecom-paristech.fr

Abstract

Nous assistons aujourd'hui à un développement continu et rapide du Web Structuré, proposant des pages générées de façon automatique, à l'aide de formes prédéfinies, et contenant des données partageant les mêmes schémas. Le passage à des données structurées ouvre de nouvelles perspectives pour la recherche d'information dans le web et soulève de nouveaux défis auxquels nous pouvons fournir des réponses. En particulier, la recherche d'information orientée données, dans laquelle l'utilisateur décrit les objets recherchés devient possible pour le web structuré. Nous proposons dans cet article `ObjectRunner`, un système pour l'extraction et l'interrogation de données du web structuré qui exploite la redondance du web et la régularité des structures de pages pour mieux déterminer les données à extraire et fournir un résultat le plus complet possible. Notre système permet à l'utilisateur de décrire de façon souple et précise le schéma des objets recherchés. Ensuite, le schéma cible et la structure des pages sources sont analysés pour déterminer les données répondant à la requête de l'utilisateur, intégrer les données pouvant correspondre aux mêmes entités du monde réel, et les extraire. La solution proposée par notre système est générique, dans le sens ou elle n'est pas spécifique à un domaine d'application ou un type d'objets en particulier. Nous montrons dans cet article son utilité, à l'aide d'exemples concrets, et son efficacité, en la comparant aux principales approches existantes.

Mots-clefs. Extraction non-supervisée, Web structuré, intégration, annotation sémantique, objets complexes.

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 are witnessing 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.

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).

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 [2, 11, 28]. Their main 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 and 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 and lightweight by enabling the elimination of unnecessary computations.

System overview. A high-level view of the `ObjectRunner` system is illustrated in Figure 1 (it will 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 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.

In this paper we will focus mainly on the technical aspects of the extraction phase and on the features of the SOD specification module. Beyond these aspects, there are other exciting research problems we are currently investigating in this system. 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 paper, whose main purpose is to report on our preliminary results on the extraction of complex objects and, more generally, 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

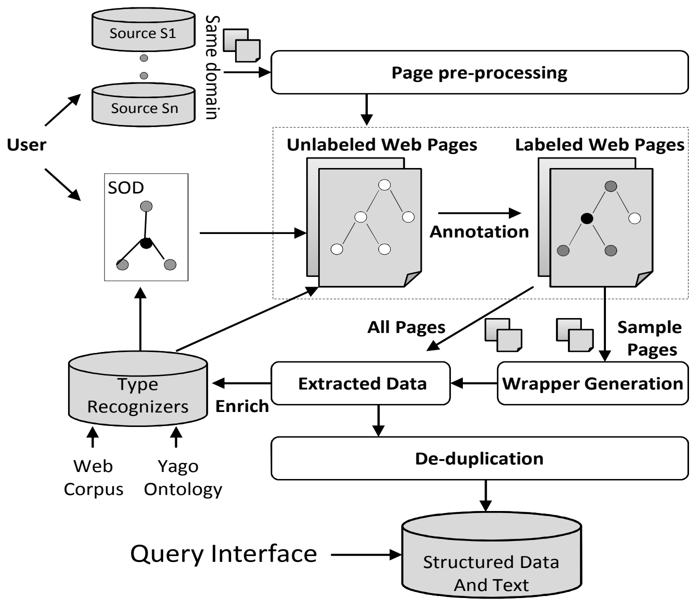


Figure 1: Architecture of the ObjectRunner system.

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.

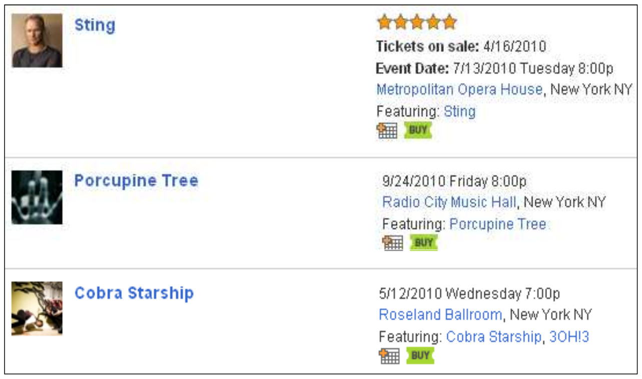
The rest of the paper is organized as follows. In the next section we introduce the necessary terminology and technical background. We provide in Section 3 a more detailed description of the system's internal structure in terms of composing parts and implementation approaches. We also illustrate how it operates through several examples. Our experimental evaluation of the system is presented in Section 4. In Section 5 we discuss the most relevant related works and we conclude in Section 6.

2 Preliminaries

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

Schematized, data-rich pages are built by retrieving information from an underlying databases. For the purpose of extraction, we identify two types of such pages. List pages encode a list of similar records, as the `zevents.com` page illustrated in Figure 2(a), which displays details on several concerts. In such pages, both high-level structure and individual records are formatted using the common template. Single-record pages focus on a single object (see example of Figure 2(b)) and, while structured, are more verbose. Very often, both kinds of pages appear in the same Web site. The former serves information in a distilled manner (e.g, our example page gives only the name of an artist, a date and an address), while the latter complement list pages by giving more details (e.g., a concert description). Even though dealing with the same data and appearing within the same Web site, the two kinds of pages will be in general considered as representing different sources, given that they rely on different templates to encode information. And depending on the targeted SOD and the level of detail it describes, listed records may suffice or not. For example, in `zevents.com` list pages the information on each concert is usually quite brief. However, if an application also needs the music concert description, one has to extract it from a detail page.

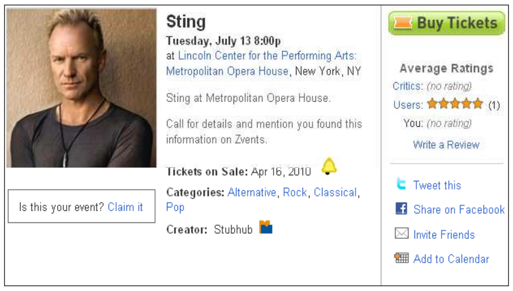
As an input for the extraction system, we suppose that the user has collected a number of structured Web sources, denoted $\{S_1, \ldots, S_n\}$, where each source represents a set of `HTML` pages that describe real-world objects (e.g., concerts, real-estate ads, books, etc). Our running example refers to `concert` objects, which can be seen as triples date-address-artist. We illustrate in Figure 3 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.


Sting
Tickets on sale: 4/16/2010
Event Date: 7/13/2010 Tuesday 8:00p
Metropolitan Opera House, New York NY
Featuring: Sting
BUY

Porcupine Tree
9/24/2010 Friday 8:00p
Radio City Music Hall, New York NY
Featuring: Porcupine Tree
BUY

Cobra Starship
5/12/2010 Wednesday 7:00p
Roseland Ballroom, New York NY
Featuring: Cobra Starship, 3OH!3
BUY

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


Sting
Tuesday, July 13 8:00p
at Lincoln Center for the Performing Arts: Metropolitan Opera House, New York, NY

Sting at Metropolitan Opera House.

Call for details and mention you found this information on Zvents.

Tickets on Sale: Apr 16, 2010
Categories: Alternative, Rock, Classical, Pop
Creator: Stubhub

Is this your event? Claim it

Buy Tickets
Average Ratings
Critics: (no rating)
Users: ★★★★★ (1)
You: (no rating)
Write a Review
Tweet this
Share on Facebook
Invite Friends
Add to Calendar

(b) A segment of a detail page

Figure 2: Structured pages from zvents.com

2.1 Types and object description

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, 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 by *isInstanceOf* recognizers). 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, 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.

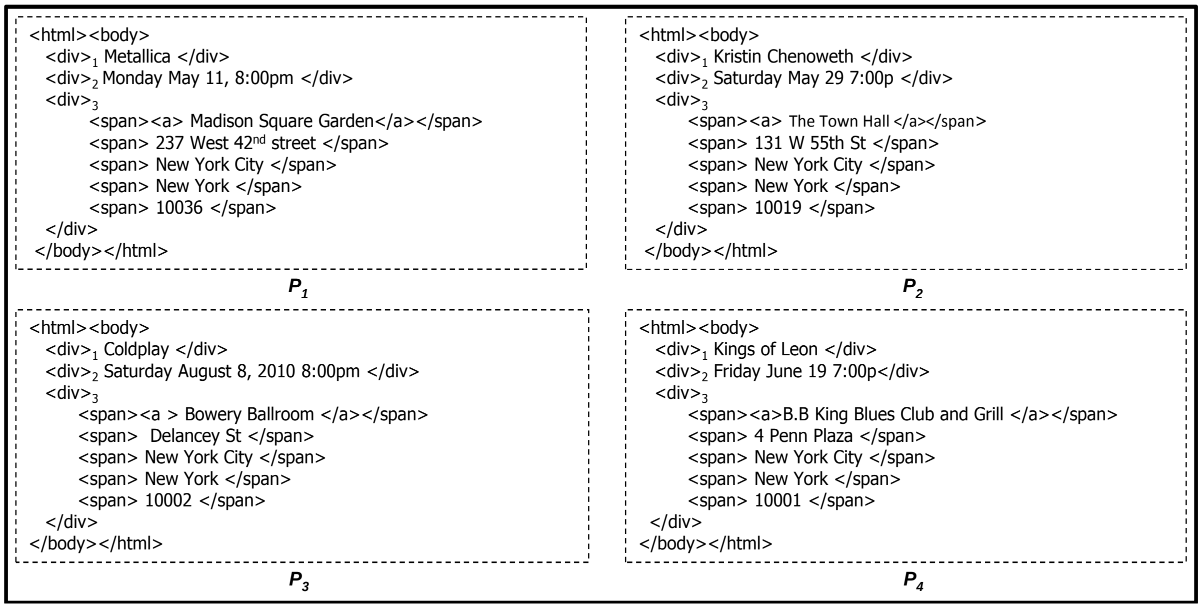
A *Structured Object Description* denotes any complex type, possibly complemented by additional restrictions in the form of value, textual or disambiguation rules. For instance, these would allow one to say that a certain entity type has to cover the entire textual content of an `HTML` node or a textual 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 entity types would be associated to predefined recognizers (for adresses and dates respectively), since this kind of information has easily recognizable representation patterns, while the last one would have an *isInstanceOf* recognizer. All the components can have multiplicity 1.

2.2 The extraction problem

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.



```
<html><body>
  <div>_1 Metallica </div>
  <div>_2 Monday May 11, 8:00pm </div>
  <div>_3
        <span><a> Madison Square Garden</a></span>
        <span> 237 West 42nd street </span>
        <span> New York City </span>
        <span> New York </span>
        <span> 10036 </span>
  </div>
 </body></html>
```

P_1

```
<html><body>
  <div>_1 Kristin Chenoweth </div>
  <div>_2 Saturday May 29 7:00p </div>
  <div>_3
       <span><a> The Town Hall </a></span>
       <span> 131 W 55th St </span>
       <span> New York City </span>
       <span> New York </span>
       <span> 10019 </span>
  </div>
 </body></html>
```

P_2

```
<html><body>
  <div>_1 Coldplay </div>
  <div>_2 Saturday August 8, 2010 8:00pm </div>
  <div>_3
       <span><a > Bowery Ballroom </a></span>
       <span>  Delancey St </span>
       <span> New York City </span>
       <span> New York </span>
       <span> 10002 </span>
  </div>
</body></html>
```

P_3

```
<html><body>
  <div>_1 Kings of Leon </div>
  <div>_2 Friday June 19 7:00p</div>
  <div>_3
       <span><a>B.B King Blues Club and Grill </a></span>
       <span> 4 Penn Plaza </span>
       <span> New York City </span>
       <span> New York </span>
       <span> 10001 </span>
  </div>
</body></html>
```

P_4

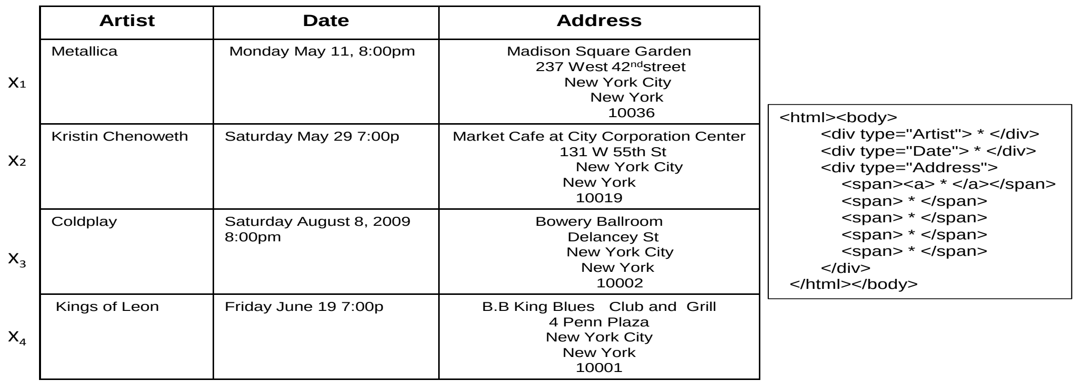
Figure 3: Sample pages

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.

We give in Figure 4 the extraction template that would be inferred in our example.

As argued in Section 1, 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:



	Artist	Date	Address
x_1	Metallica	Monday May 11, 8:00pm	Madison Square Garden 237 West 42ndstreet New York City New York 10036
x_2	Kristin Chenoweth	Saturday May 29 7:00p	Market Cafe at City Corporation Center 131 W 55th St New York City New York 10019
x_3	Coldplay	Saturday August 8, 2009 8:00pm	Bowery Ballroom Delancey St New York City New York 10002
x_4	Kings of Leon	Friday June 19 7:00p	B.B King Blues Club and Grill 4 Penn Plaza New York City New York 10001

```
<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>
```

Figure 4: The correct solution (objects and wrapper) on the running example

- *Avoiding to mix different types of information.* By relying on type recognizers to annotate input pages, we can improve the extraction process, hence 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.* In the process of building the template, if the collection of pages does not seem relevant enough as a source of structured objects, the process of wrapper inference can be stopped early.

- *Avoiding the loss of useful information.* Data that may be relevant and 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 can be 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, enriching in this way the dictionaries.

3 Overview of our approach

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 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. There are many segments in Web pages that are not data-centric, such as header information, 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, we perform cleaning of the `HTML` before the extraction process. 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 build up an algorithm that performs page segmentation (similar to VIPS [7]). We use the recognizers of the input SOD and carefully designed heuristics to chose the best candidate segment. Also, because `HTML` documents are often not well-formed, we use the open source software JTidy [17] 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/`.

3.1 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 [26], 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 `Group` class, which is itself related to `Artist` one by a *isMemberOf* relationship. For our purpose, 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 [13] 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.

3.2 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 (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). 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. The result will be a type-annotated `DOM` tree.

The top pages w.r.t. 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. For instance, in page p_1 of our example, the first `<div>` tag contains an artist name, so it will be annotated accordingly, as in `<div type=‘‘Artist’’> Metallica </div>`.

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 $S$
```

The annotations will be propagated upwards in the DOM tree to ancestors as long as these nodes have only one child (i.e., on a linear path).

3.3 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 [2].

In short, 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 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 each of the four pages of our running example, and this would correspond (initially) to the following vector of occurrences: $< 3, 3, 3, 3 >$. 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 `ExAlg` this depended on two criteria: the position in the DOM tree of the page and the position w.r.t. 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 (either not properly ordered or not nested) are discarded.

How roles and equivalence classes are computed distinguishes our approach from [2]. 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 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.

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:         differentiate roles using EQs + non-conflicting annotations
7:     until fixpoint
8:     differentiate roles using EQs + conflicting annotations
9: until fixpoint
```

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

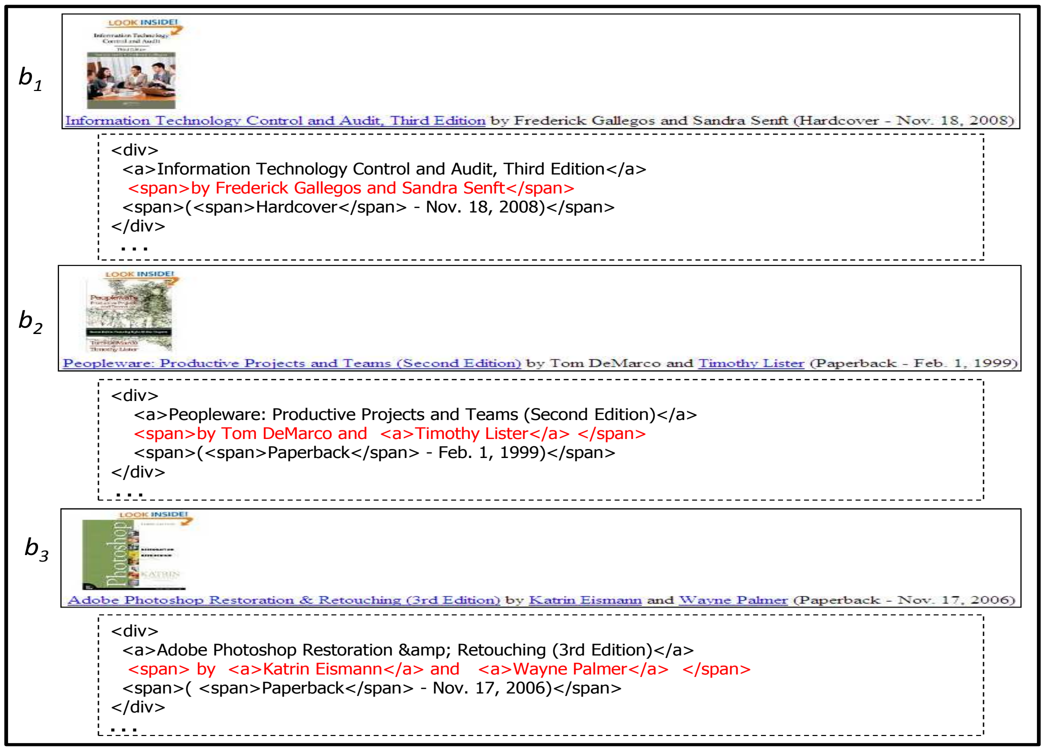
- **Non-conflicting annotations.** A token - identified by its DOM position and its coordinates w.r.t. the existing equivalence classes - has non-conflicting annotations if each of its occurrences have the same (unique) type annotation or no annotation at all.

- **Conflicting annotations.** A token has conflicting annotations different type annotations have been assigned to its occurrences.

In the first case, tokens with no conflicting annotations are treated in the loop along with the other criteria. Once all equivalences classes are computed in this way, we carry out one additional iteration in order to find new occurrence vectors and equivalence classes. The entire process is then repeated until a fix-point is reached.

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 `<div>`

Metallica Monday May 11, 8:00pm	Madison Square Garden	237 West 42nd street	10036



Information Technology Control and Audit, Third Edition by Frederick Gallegos and Sandra Senft (Hardcover - Nov. 18, 2008)

```
<div>
  <a>Information Technology Control and Audit, Third Edition</a>
   <span>by Frederick Gallegos and Sandra Senft</span>
  <span>(<span>Hardcover</span> - Nov. 18, 2008)</span>
</div>
 . . .
```

Peopleware: Productive Projects and Teams (Second Edition) by Tom DeMarco and Timothy Lister (Paperback - Feb. 1, 1999)

```
<div>
    <a>Peopleware: Productive Projects and Teams (Second Edition)</a>
    <span>by Tom DeMarco and  <a>Timothy Lister</a> </span>
    <span>(<span>Paperback</span> - Feb. 1, 1999)</span>
</div>
 . . .
```

Adobe Photoshop Restoration & Retouching (3rd Edition) by Katrin Eismann and Wayne Palmer (Paperback - Nov. 17, 2006)

```
<div>
  <a>Adobe Photoshop Restoration &amp; Retouching (3rd Edition)</a>
   <span> by  <a>Katrin Eismann</a> and   <a>Wayne Palmer</a>   </span>
  <span>( <span>Paperback</span> - Nov. 17, 2006)</span>
</div>
 . . .
```

Figure 5: List page with three book records from Amazon

occurrences would have the same role. This would lead for instance to the following (incorrect for our purposes) extraction result from page P_1 shown in the table below.

For further illustration, there are other situations where the annotations improve the result of the extraction. For example, the `amazon.com` page fragment in Figure 5 shows a list of 3 books (indicated as b_1, b_2 and b_3). A book can have one or more authors, and they are represented differently in `HTML` tree. Taking into account only the positions of tokens in equivalence classes can result in extracting author names as values in distinct fields of the template. But with the annotation of the tag `<span>` we can determine that this attribute represents author names and extract it 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 `<div>` had the same number of occurrences in all pages, 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).

3.4 Querying the extracted data

We designed a query interface that (i) enables users to define SODs, and (ii) query the extracted data. In the SOD specification phase, users can either build on existing SODs or specify new types along with means to recognize them (e.g., using the `Yago` ontology). For querying extracted data, one choses which SODs will be used to query the Web. Then, 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.

4 Experimental Results

The experiments have been performed[1] on three different domains, with both new datasets and datasets that have been already used in other wrapper inference experiments. They consist of both list and detailed pages collected from surface or hidden Web sources.

The pages refer to the following three domains: `concerts`, `books` and `albums`. We randomly selected 100 pages per source from Web sites such as `zvents.com` and `upcoming.yahoo.com` for concerts, `amazon.com` and `bn.com` for books[2]. For each domain, we have collected 10 sources that were matched with one SOD as described bellow:

- **Concerts.** A `concert` object is composed of three entity types: `artist`, `date` and `location`. For each type, the multiplicity constraint is set to 1 ($m_i = 1$).

[1]We used a commodity workstation equipped with a 2.8 GHz CPU, 3GB of main memory and the Java programming environment.

[2]Most of the sites used for the books and albums domains were used in the TurboSyncer [10] experiments

- **Books.** A `book` object is composed of four entity types: `title`, `author`, `price` and `date`. The multiplicity constraint has been set to 1 for the first three entity types, while the fourth one is optional.
- **Albums.** An `album` object is composed of four entity types: `title`, `artist`, `price` and `date`. The multiplicity constraint has been set to 1 for the first three entity types, while the fourth one is optional.

We compared our algorithm `ObjectRunner` (OR) with two of the most cited and closely related works, namely `ExAlg` [2] (EA) and `RoadRunner` [11] (RR). The goal of these experiments was to (i) evaluate the robustness of our system, and (ii) compare our results with the ones of the existing algorithms.

The selected 100 pages from each source are assumed to have the same template. We show in Table 1 the precision of template construction - i.e., how many components 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 existed or not in the given source (we denote this in the table). We then manually checked the results and classified them as follows:

- **Correct attributes**. An attribute is classified as correct if the extracted values for it are correct.
- **Partially correct attributes**. An attribute is classified as partially correct if: (1) the values for two or several attributes are extracted together (as instances of one field) and they appear in the same manner in pages (for example, a book title and the author name may in the same text as an atomic piece of information), or (2) the source is made of list pages and listed values corresponding to the same attributed of the SOD are extracted separately.
- **Incorrect attributes**. An attribute is classified as incorrect if the extracted values are incorrect, i.e., a mix of values corresponding to distinct attributes of the implicit schema.

In Figure 6, we compare the results of `ObjectRunner` with those obtained by the two other algorithms, in order to quantify the effect of the semantic annotations (the SOD) on the extraction process. For 10 sources per domain, we provide several facets for accuracy: (1) 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), (2) the rate of correct, partially correct and incorrect attributes, and (3) the rate of sources that were incompletely handled (i.e., with partially correct or incorrect attributes).

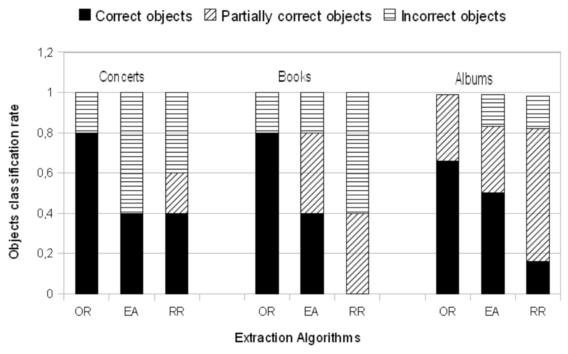
Domains	Sites	Attributes	Optional	Correct	Partially	Incorrect
concerts	zvents.com	3	yes	3	0	0
	upcoming.yahoo.com		yes	3	0	0
	eventful.com		yes	1	1	1
	bandsintown.com		yes	3	0	0
	stubhub.com		yes	3	0	0
books	amazon.com	4	yes	4	0	0
	bn.com		yes	4	0	0
	buy.com		no	3	0	0
	abebooks.com		no	3	0	0
	walmart.com		yes	3	0	1
albums	amazon.com	4	yes	3	0	1
	buy.com		yes	4	0	0
	101cd.com		no	1	2	0
	towerrecords.com		yes	4	0	0
	walmart.com		yes	3	1	0

Table 1: Extraction results

In Figure 6(b) we detail the results that were summarized in Figure 6(a). More precisely, for each SOD, we give the rate of correct, partially correct and incorrect attributes. Note that the results are symmetrical to those observed in Figure 6(a). In this view of the results as well, our algorithm has better results than the existing approaches.

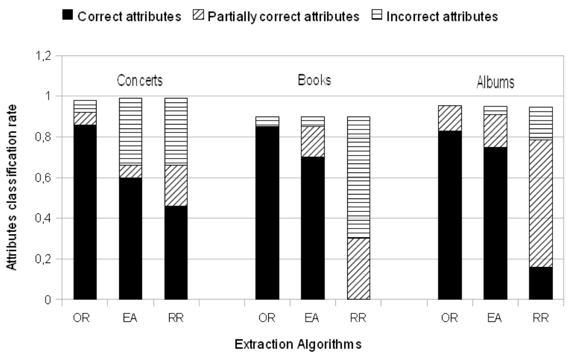
Figure 6(c) compares the three algorithms by their ability to handle a source correctly. We observe that in all the three domains OR outperforms EA and RR. Indeed, there are 20% incompletely handled sources in the first two domains and 30% in the third one.

As a final remark, as Web data tends to be very redundant, it becomes important to be able to handle correctly at least some sources from a given domain than to handle many sources in fair but incomplete (partially) manner. For example, 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`.



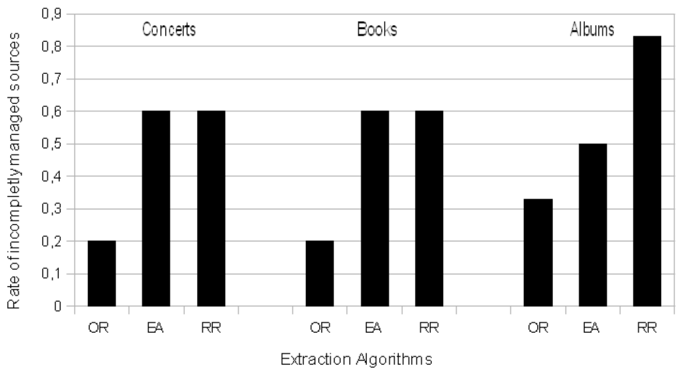
OR: ObjectRunner EA: ExAlg RR: RoadRunner

(a) Objects classification



OR: ObjectRunner EA: ExAlg RR: RoadRunner

(b) Attributes classification



OR: ObjectRunner EA: ExAlg RR: RoadRunner

(c) Incompletely managed sources

Figure 6: ObjectRunner comparison

5 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 [18]). The manual approaches extract only the data that the user marks explicitly, using either wrapper programming languages, such as the ones proposed in [14, 25], or visual platforms to construct extraction programs, like WICCAP [20], Wargo [23] or Lixto [12]. Supervised approaches use learning techniques, called wrapper induction, to learn extraction rules from a set of manually labeled pages (WIEN [19], XWrap [21], Softmealy [16], Stalker [22]). 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 [8], Thresher [15]) arrive to reduce human intervention by acquiring a rough example from the user. Some semi-supervised approaches (such as IEPAD [9]) 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 [27], DEPTA [28]), which is often not the case in practice. RoadRunner [11], 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 [2] 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 [10] is an integration system which can incorporate many sources and uses existing extraction results to better calibrate future extractions.

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 [5], Snowball [1], KnowItAll [6]), or by discovering relations automatically (TextRunner [4]). Other systems in this area, like Yago [26] and DBpedia [3], extract relation by rule-based harvesting of facts from semi-structured sources such as Wikipedia. For a survey on the management of information extraction systems see [24].

6 Conclusion

This paper considers an alternative approach to automatic information extraction and integration from structured Web pages. It advocates the advantages of two-phase querying, 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 three different domains) and the source structure, our system harvests more real-world items, with fewer errors.

Besides the extraction tasks, there are other exciting research problems we are currently investigating in the `ObjectRunner` system. For instance, we are studying techniques for discovering, processing and indexing structured Web sources. Also, given an input SOD, we would like to be able to automatically select the most relevant and data rich sources.

Finally, our wrapper inference technique based on equivalence classes for page tokens is amenable to parallel execution, while the extraction step itself can obviously be performed in parallel. We are currently investigating the implementation of our algorithms in a distributed shared-nothing architecture.

References

[1] E. Agichtein and L. Gravano. Snowball: extracting relations from large plain-text collections. In *DL Conference*, 2000.

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

[3] 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.

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

[5] S. Brin. Extracting patterns and relations from the world wide web. In *WebDB*, 1998.

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

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

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

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

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

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

[12] 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.

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

[14] A. Hemnani and S. Bressan. Extracting information from semi-structured web documents. In *OOIS Workshops*, 2002.

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

[16] C.-N. Hsu and M.-T. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Inf. Syst.*, 1998.

[17] JTidy. http://jtidy.sourceforge.net.

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

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

[20] Z. Li and W. K. Ng. WICCAP: From semi-structured data to structured data. In *ECBS*, 2004.

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

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

[23] J. Raposo, A. Pan, M. Álvarez, J. Hidalgo, and A. Vi na. The wargo system: Semi-automatic wrapper generation in presence of complex data access modes. In *DEXA*, 2002.

[24] S. Sarawagi. Information extraction. *Foundations and Trends in Databases*, 2008.

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

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

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

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