Automatically Maintaining Wrappers for Semi-Structured Web Sources

Juan Raposo, Alberto Pan, Manuel Álvarez
Department of Information and Communication Technologies. University of A Coruña.
{jrs,apan,mad}@udc.es

Justo Hidalgo
Denodo Technologies Inc.
jhidalgo@denodo.com

ABSTRACT
In order to let software programs gain full benefit from semi-structured web sources, wrapper programs must be built to provide a “machine-readable” view over them. A significant problem in this approach arises as Web sources may undergo changes that invalidate the current wrappers. In this paper, we present novel heuristics and algorithms to address this problem. In our approach the system collects some query results during normal wrapper operation and, when the source changes, it uses them as input to generate a set of labeled examples for the source which can then be used to induce a new wrapper.

1 INTRODUCTION

1.1 Semi-structured Web sources

A substantial subset of Web data follows some kind of underlying structure. For instance, the pages obtained as a response to a query executed through a Web search form are usually generated by a program that accesses structured data in a local database and embeds them in HTML templates. These Web sources are usually called semi-structured sources.

In many cases, semi-structured Web sources contain highly valuable information that cannot be easily accessed by other means. For instance, data may be provided by an autonomous organization that only provides Web access for its information. On other occasions, directly accessing the back-end database may be difficult or inconvenient.

1.2 Web wrappers

A program that is able to provide software applications with a structured view of a semi-structured Web source is usually called a wrapper. Wrappers receive queries from calling applications and automatically execute them on the source. Answering a query usually involves automatically filling in some Web form and extracting the data from the returned Web pages, thus delivering a set of structured results to the calling application. For instance, Fig. 1 outlines the process followed by a wrapper to execute a certain query on an Internet bookshop.

Since wrappers enable applications to access Web data in a similar manner to that of information from databases, they are a key part of most current Web data integration architectures such as mediator systems [13].
Automatic and semi-automatic generation of wrappers has been an active research field for some years. Most works have focused on the step 3 of Fig. 1: the problem of extracting structured data from HTML pages [4,6,7,12] (see [8] for a survey).

An approach to this problem of particular interest to our case is called *wrapper induction* [4,6,7]. In this approach the wrapper creator manually labels some data elements in some sample pages from the source. These data elements are then used as examples by the system to automatically learn how to extract the desired data from other pages following the same (or a slightly different) template.

Fig. 2 sketches the process needed to generate a wrapper able of extracting data from an Internet bookshop using the wrapper induction approach.

---

**Fig. 1. Wrapper Execution**

1. **Application**
   - The application asks a query \( q = \text{TITLE contains XML} \)

2. **Wrapper automatically fills a form and submits the query**

3. **Wrapper extracts structured body information and returns it to application**

**Extracted Data**

- **Title: XML in a Nutshell**, Third Edition
  - Author: Elliotte Rusty Harold, W. Scott Means
  - Format: Paperback
  - Date: September 2004
  - Price: $26.37

- **Title: Beginning XML**, Second Edition
  - Author: David Hunter, et al
  - Format: Paperback
  - Date: March 2003
  - Price: $27.19
1.3 Automatically maintaining wrappers

The main problem with wrappers is that they usually stop working properly when the Web sources change the underlying templates of their pages. This is called the maintenance problem. In this paper, we deal with this issue.

Our approach for automatic maintenance is based on collecting some of the query results obtained during wrapper operation. When the source changes, these results are used by a novel algorithm based on a series of heuristics, to generate a new set of labeled examples that can then help to generate a new wrapper using induction techniques (i.e. once they have been labeled, the collected results act as user-provided examples during the initial wrapper generation). Fig. 3 sketches the process stages.

Thus, the automatic maintenance process comprises the following four stages in our approach:
1) Collecting results. While the wrapper answers queries, it stores the results from some of them.

2) Wrapper verification. The system periodically checks whether the wrapper is still valid or not.

3) Generating a new training set. When it has been determined that the wrapper is no longer valid, the system must generate a new set of labeled examples to regenerate the wrapper. Having as input the results stored in stage 1 and the web pages that constitute
the current response in the source to the stored queries, this stage outputs a set of examples labeled in the response pages.

4) Regenerating wrappers. From the examples generated in the previous stage the new extraction rules can be reinduced.

In this paper we will focus on the first and third stages. Methods for stage 2 have been addressed in [5,9]. In stage 4, we use our own induction algorithm (see section 4) although our implemented system could also be easily adapted to another one.

1.4 Organization of the paper

The rest of the paper is organized as follows. Section 2 describes the methods used to select the query results that must be stored during wrapper operation. Section 3 is the core of the paper and contains our main contributions: it presents the novel heuristics and algorithms that constitute our approach for generating the new set of examples from the stored query results. Section 4 overviews our wrapper induction algorithm for regenerating the wrapper from the new set of examples. We have experimented with our techniques in several important real-world Web data extraction problems and have found them to be highly accurate. These experiments are described in section 5. Section 6 discusses related work, and section 7 summarizes our conclusions.

2 COLLECTING RESULTS FROM QUERIES

The objective of this stage is to collect the results from a subset of the queries executed by the wrapper during its normal operation so they can be used to generate a new training set of labeled examples when the source changes. The system works as follows:

1) When the wrapper answers a query, it stores up to a certain (customizable) number of results from the query in a local database. For each result collected, the system stores the following
data (the use of which will be described later): the query that generated it, the set of extraction rules used by the current wrapper to parse it and an expiration date computed from the current time.

2) The current content in the database is explored periodically to check whether the non-expired results are sufficient to satisfy certain requirements we will describe below or not. If so, all the expired results are removed. If not, then the system computes a minimum set of expired queries that must remain stored.

The requirements we want the stored results to satisfy are:

1) Storing at least a certain (customizable) number of query results. In general, the regeneration process will provide better output results when using a sufficient number of results. Our current implementation tries to store at least 100 results.

2) Storing results from at least a certain (customizable) number of different queries. Queries may implicitly determine some properties of the response pages. For instance, an electronic shop may assign a given discount (e.g. 15%) to all the books by a given author \( a \). If we used results only from the query ‘\( \text{author}=a \)’, the wrapper induction process could wrongly consider the string “Special discount: 15%” to be present in all the pages from the source, and this could lead to the generation of incorrect extraction rules for the new wrapper. Our current implementation stores results from at least 10 different queries.

3) Storing a minimum (customizable) number of results for each current visual representation of a result in the source. Most Web sources use slightly different visual representations for the results of a query. For instance, a book may include some discount information which another book does not, and that may affect the way it is shown on the page. The wrapper will usually need a different extraction rule (or set of extraction rules) to extract data items using each
visual representation. So by storing the extraction rule or set of extraction rules used by the current wrapper to obtain a result, we can tell if two query results have the same visual representation in the source or not. When the source changes, the wrapper regeneration process will usually benefit from having examples of each visual representation in order to generate the extraction rules needed to parse all the data items. We have heuristically found it is very useful to store at least a minimum number of results using each former visual representation (before the source change) in order to increase the probability that the stored results also cover all the new visual representations (after the source change). In our implementation we have set this value to 5.

3 GENERATING A NEW SET OF LABELED EXAMPLES

In this section we describe our approach for obtaining new labeled examples from the stored query results. It is organized as follows: section 3.1 introduces the data model underlying our system. Section 3.2 states the labeling problem and explains some of its difficulties. Section 3.3 describes the basic heuristics and algorithms of our approach for the base case where the data to be extracted may be modeled as a type exclusively composed of atomic fields. Section 3.4 generalizes those techniques to deal with types of arbitrary depth.

3.1 Data model

In this section we describe the data model we use to represent the results returned as response to a query on a semi-structured web source. We consider each query result as belonging to a type. Every type has an associated name and is composed by a list of fields. Each field also has an associated name. The value of a field is defined recursively as follows:

- A field can be atomic. In that case, its values are character strings.
If \( field_1, ..., field_n \) are fields, then we can define a new compound field \( T[field_1, field_2, ..., field_n] \). A value for \( T \) is a set whose elements are ordered lists of values \( v_1, ..., v_n \), where \( v_i \) is a value of \( field_i \).

For instance we can model the schema of the data elements of a type called \( ALBUM \) as

\[
ALBUM = \{TITLE, ARTIST, DATE, EDITION\{FORMAT, PRICE\}\},
\]

where \( TITLE, ARTIST \) and \( DATE \) are atomic fields and \( EDITION \) is a compound field defined upon the atomic fields \( FORMAT \) and \( PRICE \).

A type can be seen as a tree whose leaves are atomic fields and the non-leaf nodes represent non-atomic fields. Fig. 4 shows the tree of the \( ALBUM \) type along with some data instances of it.

Fields which are not direct descendants of the root will be usually referred to using its complete path in the tree. For instance, in our example we would refer to \( FORMAT \) and \( PRICE \) fields as \( EDITION.FORMAT \) and \( EDITION.PRICE \).

### 3.2 The problem of correctly labeling the examples

Now we proceed to describe the problem of generating a new set of labeled examples. At the labeling stage, we have the following inputs:

1) A source modeled according to a type \( T = \{field_1, ..., field_n\} \).

2) The set of queries \( Q = \{q_1, ..., q_m\} \) whose results were collected during wrapper operation.
3) For each \( q_i \in Q \), we will use its stored results as input. We will consider them a set of unlabeled examples (or examples for short) \( E_i = \{ e_{i1}, \ldots, e_{im} \} \).

4) For each \( q_i \in Q \), the page or set of pages \( P_i \) that form the current response (after the change) to \( q_i \) in the source.

The objective of this stage is to label the maximum possible number of examples in the new pages. Labeling an example \( e_{ij} \) consists in locating it in \( P_i \) by correctly identifying the values of all its constituent data fields.

More formally, we can define the problem of correctly labeling an example in this case as follows. Let us assume we have an example \( e \) of type \( T = \{ \text{field}_1, \ldots, \text{field}_r \} \) which we want to label on page \( P \). For the sake of simplicity, let us also assume that all the fields of \( T \) are atomic (section 3.4 will explain how to deal with data types of any complexity).

Examples of type \( T \) are of the form \( e = \{(\text{field}_1, \text{value}_1), \ldots, (\text{field}_r, \text{value}_r)\} \), where \( \text{value}_k \in \{1..r\} \), is a string denoting the value for the \( k \)-th field of \( T \) in \( e \). Each string \( \text{value}_k \in \{1..r\} \) may appear multiple times in the set of pages \( P \), and we will term each appearance of \( \text{value}_k \) as a candidate field occurrence of \( \text{field}_k \) for \( e \).

A candidate example occurrence for \( e \) in \( P \) is of the form \( c_e = \{(\text{value}_1, \text{pos}_1), \ldots, (\text{value}_r, \text{pos}_r)\} \), where \( (\text{value}_k, \text{pos}_k) \in \{1..r\} \) denotes the \( \text{pos}_k \)-th occurrence in \( P \) of the string \( \text{value}_k \).

A candidate example occurrence \( c_e \) will be a correct labeling for \( e \) if the identified field occurrences for all the \( \text{value}_k \in \{1..r\} \) form in \( P \) the real appearance of the data element represented by \( e \).

For instance, let us consider the case of maintaining a wrapper for an electronic bookshop whose results are of type \( T = \{ \text{TITLE}, \text{AUTHOR}, \text{FORMAT}, \text{PRICE} \} \), where \( \text{TITLE}, \text{AUTHOR}, \text{FORMAT} \) and \( \text{PRICE} \) are atomic. Let us also suppose an example \( e_{ij} = \{(\text{TITLE}, \text{"Beginning} \} \).

Fig. 5. Candidate occurrences
Fig. 5 shows a page containing several candidate occurrences of \(e_{ij}\) and a simplified version of its HTML code (the HTML code of the real page is much larger, although the ideas we will discuss are equally applicable). We annotate the occurrences on the page of each value of fields of \(e_{ij}\) subscripted with its relative order with respect to the total number of appearances of the value on the page. For instance, the value of the title field ("Beginning XML, Second Edition") appears twice on the page, and so its appearances are marked as "Beginning XML, Second Edition\(_1\)" and "Beginning XML, Second Edition\(_2\)".

Fig. 5b shows some of the candidate occurrences of \(e_{ij}\) on the page. Our objective is to choose the correct one among all the possible candidates. Candidate occurrences 2 and 4 in our example are incorrect, since they are formed of values from different data instances. For instance, candidate occurrence 4 takes the price value from the book titled ‘Learning XML, Second Edition’, whereas the values for the remaining fields are taken from the book ‘Beginning XML, Second Edition’.

It could seem that both occurrences 1 and 3 are correct. Occurrence 1 corresponds with the appearance of the desired book on the list of ‘most popular results’, whereas occurrence 3 corresponds with its appearance on the list of complete results. Nevertheless, since we want our examples to help generate extraction rules to extract all the results and not only the most popular ones, we should use occurrence 3.

Another complication arises because the stored example might actually not have any correct candidate occurrence on the page, even if some candidate occurrences exist. For instance, Fig. 6 shows a sample page where there are some candidate occurrences for \(e_{ij}\), but none of them is correct. Therefore, our system needs to detect these situations and discard these “fake” examples in order to provide the re-induction stage only with valid examples.
Since there will be some time interval between the moment the results are collected and the moment we try to regenerate the wrapper (this time interval will not usually be larger than a few hours since automated verification methods can be used to detect the change), the data on the source may have changed during that interval. Note that the main assumption of our approach is that we can usually be confident that a sufficient number of unlabeled examples contained in $E_i$ (that is, the stored results for the query) is still present in $P_i$ after the change in the source. It should be noted that in those cases where the values of the data elements to be extracted from the source vary very rapidly (e.g. stock quotes information) this assumption can be wrong. In section 3.3.5 we will describe how our approach can be extended to deal with those sources.
### 3.3 Labeling examples of types composed of atomic fields

This section describes the basic heuristics and algorithms of our approach for the base case where the data to be extracted may be modeled as a type exclusively composed of atomic fields.

As we have already said, we have as inputs a source modeled according to a type \( T \) along with the set of queries \( Q \) whose results were collected during wrapper operation. For each \( q \in Q \), we will have a set of unlabeled examples \( E = \{ e_1, \ldots, e_n \} \) and the page or set of pages \( P \) that constitute the current answer to \( q \) after the change in the source.

The pseudo-code of the algorithm to label an example \( e_i \in E \) in \( P \) is shown in Fig. 7.
Overall, the algorithm works as follows (the details will be provided in the following sub-sections). It starts by computing all the candidate example occurrences in $P$ for $e_i$. If their number is higher than a certain (customizable) limit, then the most promising ones are chosen. Next, the algorithm ranks them according to three heuristics: the “sibling nodes” heuristic, the “pattern” heuristic and the “proximity” heuristic. The “sibling nodes” and “pattern” heuristics are both based on what we term the “list” observation: both heuristics try to rank higher the candidate example occurrences that are part of a list of items with a similar layout on the page and where some of the items on the list match other examples from $E$. The “proximity” heuristic ranks candidate example occurrences according to the proximity on the page of the data values that form them.

The algorithm uses the “sibling nodes” heuristic as the main criterion for choosing the best candidate occurrence. Nevertheless, there are some cases where it is not applicable. In that case, the “pattern” heuristic is used as the main criterion. The “proximity” heuristic is used only when the main criterion returns the same value for several candidates. As we will see, this is because the “sibling nodes” and “pattern” heuristics take into account the context provided from all the other examples, thus making them very effective.

The following sub-sections describe in detail several aspects of the algorithm: section 3.3.1 details the “list” observation and the heuristics we have defined based on it: the sibling nodes and pattern heuristics. Section 3.3.2 describes the proximity heuristic. Section 3.3.3 addresses the problem of preselecting candidate occurrences when their number is too high to compute the heuristic values for all of them. Section 3.3.4 deals with the detection of “fake” examples. Finally, section 3.3.5 describes the extensions needed to our approach to deal with the case of sources having data fields that vary very rapidly in time.
3.3.1 The “List” Observation

The “list” observation states that the data elements obtained as a response to a query in a semi-structured Web site are usually instances of a list whose elements share a similar layout.

So if a given candidate occurrence $c$ for an example $e_i \in E$ is the correct occurrence, then it is very probable that the page also contains other examples from $E$ with a similar layout to $c$ and arranged on the page as a list.

It is important to note that this observation also applies when the data elements are distributed on several pages. For instance, if each page contains only one data instance (e.g. the detail page for each book in an electronic bookshop), the observation still applies by simply considering a single “aggregated page” containing all the original pages.

We have developed two heuristics based on the list observation: the sibling nodes heuristic and the pattern heuristic. They are described in the following sub-sections.

3.3.1.1 Sibling Nodes Heuristic

The sibling nodes heuristic is based on an additional observation: given the DOM tree of an HTML page containing a list of structured items, each item is usually represented using a sub-tree whose root node is a “sibling” of the root nodes from the sub-trees corresponding with the other items on the list. We consider two nodes as “siblings” if they have the same Xpath route from the root of the DOM tree.

For instance, Fig. 8 shows an excerpt of the DOM tree of the HTML page of Fig. 5. As can be seen, the books on the list are represented in sibling sub-trees.

Therefore, to rank a candidate occurrence $c_k$ of an example $e_i \in E$ (remember: $E$ is the set of unlabeled examples stored for a given query $q$), the sibling nodes heuristic find the deepest sub-tree containing $c_k$, and searches its sibling sub-trees for candidate occurrences of other examples.
$e_j \in E, j \neq i$ having “similar” layout as $e_i$ (we will explain what we mean by "similar" in brief).

The “ranking” for the occurrence will be the number of other examples found in the sibling sub-trees.

Let us consider the example shown in Fig. 9. The figure shows two candidate occurrences $c_1$ and $c_2$ for the example $e_1$ along with the set of stored results $E=\{e_1,e_2,e_3\}$ and the DOM tree of a page containing some results from $E$. $c_1$ is the correct occurrence of $e_1$ while $c_2$ is a fake occurrence comprising data fields from two different data elements. The sub-tree $s_I$ is the deepest sub-tree containing $c_1$. As can be seen, its sibling sub-trees $s_2$ and $s_3$ contain candidate occurrences of, respectively, $e_2$ and $e_3$, and they are disposed in their sub-trees in the same way as $e_1$ is disposed in $s_I$; consequently, the ranking for $c_1$ according to the sibling nodes heuristic is 2.
In turn, the deepest sub-tree containing \( c_2 \) is \( s_4 \), which has as root a node which is “too high” in the DOM tree and, therefore, its siblings \( (s_5, \ldots) \) do not contain candidate occurrences of any other examples; thus, the ranking for \( c_2 \) will be 0.

The algorithm used to implement the sibling nodes heuristic is shown in Fig. 10. Now we discuss some important aspects of it.

A crucial consideration is the method used in steps 1.b.ii.1 and 1.d.ii.1 to search for candidate example occurrences with a “similar layout” to \( c_k \). The basic idea is to count only those candidate occurrences that have their values positioned in the tree in a similar way to \( c_k \). To achieve this,
we perform two actions: first, we compute the Xpath route for every field occurrence of each candidate example occurrence, and we compare them with the route to the corresponding field occurrence in c_k, checking if they are equal. Secondly, we check whether the ordering of the field occurrences is the same as in c_k (e.g. if the TITLE appears before the AUTHOR in c_k, the same should happen with the candidate occurrences of the other examples).

It should also be noted that in step 1.b.ii.1, if occurrences with a “similar layout” to c_k are found for any other example in the sub-tree of c_k, then numSiblingOccurrences_k will be 0 for this candidate occurrence, and we will continue with the next one. The reason for this is illustrated in Fig. 11 and explained below. The shown candidate occurrence is incorrect, since the field occurrences that form it are taken from several distinct data items on the list. When the sibling

---

**Algorithm:** Obtains the list of candidate occurrences for an example e_i ranked according to the sibling nodes heuristic.

- RankedC_i = SiblingNodes (P, T, E, c_i, C_i)

**Input:**
- P, a document (or a set of documents considered as if they were part of a single page where their DOM trees were siblings).
- A type T = {field_1, …, field_r}.
- E = {e_1, …, e_n}, where e_j = {field_k, value_k} is an example data element of type T contained in P.
- e_i is the example contained in E whose candidate occurrences will be ranked. e_i is of the form {(field_1, value_1), (field_2, value_2), …, (field_r, value_r)}.
- C_k = {c_1, …, c_m} is the set of candidate example occurrences for e_i in P. Each c_k = {value_k, pos_k_1, pos_k_2, …, pos_k_r}, where (value_k, pos_k_p) denotes the pos_k_p-th occurrence on the page of the string value_k and, thus, implicitly identifies the node in the DOM tree of P where such occurrence begins (note that the occurrence could extend through several nodes).

**Output:**
- RankedC_i, the list of candidate example occurrences for e_i ranked according to the sibling nodes heuristic.

1) For each c_k ∈ {c_1, …, c_m}:
   a. numSiblingOccurrences_k = 0
   b. Obtain A_k, the root of the deepest subtree comprising the nodes that form c_k.
      i. For each e_j ∈ {e_1, …, e_n}, e_j ≠ e_i, search for candidate occurrences of e_j in the sub-tree with root A_k.
      ii. If candidate occurrences are found for any e_j, then
         1. If any of the candidate occurrences found has a “similar layout” to that of c_k, then rank this occurrence with zero value and continue with the next candidate example occurrence.
   c. Obtain {S_1, …, S_t} the set of sibling nodes of A_k in the DOM tree of P (note that A_k is not included in this set).
   d. For each sibling node S_p, p = 1, …, t:
      i. For each e_j ∈ {e_1, …, e_n}, search for candidate occurrences of e_j in the sub-tree with root S_p.
      ii. If candidate occurrences are found for exactly one e_j ∈ {e_1, …, e_n}, then
         1. If any of the candidate occurrences found has a “similar layout” to that of c_k, then numSiblingOccurrences_k++.
2) Sort candidate example occurrences in {c_1, …, c_m} by numSiblingOccurrences, in descending order.

Fig. 10. Sibling Nodes Heuristic Algorithm
nodes heuristic is true, then the deepest sub-tree containing such incorrect candidate occurrences will also contain several (typically all) items on the list (as shown in the figure where the subtree also contains examples e₃ and e₄). So we will probably find occurrences of several other examples, and this is an excellent indicator that the candidate occurrence is incorrect. That is why, in these cases, the heuristic ranks it with the value 0.

Finally, the step 1.d.ii of the algorithm increases \( \text{numSiblingOccurrences}_k \) if, and only if, the candidate example occurrences found in a certain sibling sub-tree belong to exactly one \( e_j \in E_{\neq i} \).

The reason for this is very similar to that of the previous consideration: if we find occurrences of more than one example in the same sibling sub-tree, then it means we are searching from a node which is “too high” and the candidate occurrence is probably incorrect, because it spans through two data items, and in that case its ranking should not improve.
When this heuristic ranks 0 for every candidate occurrence of a given example, we consider it a “fake example” and discard it without returning any correct labeled occurrence (recall step 6 in Fig. 7).

3.3.1.1 Applicability of the sibling nodes heuristic

According to our experiments the sibling nodes heuristic is very effective. Unfortunately, it is not always applicable. In some relatively rare cases the DOM tree of the page is built in such a way that there is not a unique root node for the sub-tree containing each item of the list, but instead all items share the same “deepest sub-tree”. Fig. 12 shows an example of this situation.

In these cases, even the correct candidate occurrences will rank zero in the heuristic since we will always find occurrences of several other examples in the deepest sub-tree of the occurrence (recall step 1.b.ii.1 in Fig. 10).

Therefore, when the sibling nodes heuristic returns zero for every occurrence of every example, we consider the sibling nodes heuristic not applicable and the pattern heuristic will be used.
3.3.1.2 Pattern Heuristic

This heuristic also exploits the “List” observation. In this case, to rank a candidate occurrence $c_k$ for an example $e_i \in E$ (where $E$ is the set of examples stored for a given query $q$) we compute from $c_k$ a regular expression-like “pattern” trying to capture its “layout”. The pattern will contain placeholders in the positions where the values of the example appear. Then we will apply the pattern to $P$, obtaining a set of matches, where each match is composed of one string value for every placeholder in the pattern. The basic intuition is that if the candidate occurrence is correct, then the values extracted for some of the matches will coincide with the values from other examples from $E$. The number of examples found will be the ranking assigned to $c_k$ by the heuristic.

To compute the pattern for a candidate occurrence, we traverse the tree in deep-first order from the first value of the occurrence to the last. We substitute the string values of the example by the placeholders $\$FIELD\_NAME$ and the rest of the strings by the special token $\$ANY$ (matches with every string). To generate the pattern we ignore some tags that we have found of little relevance to capture the basic layout of the occurrence such as font, span, bold or italic tags.

The pattern heuristic is illustrated in the example of Fig. 13, where we have two candidate occurrences $c_1$ and $c_2$ for the example $e_1$. $c_1$ (the correct occurrence) has been used to generate the pattern $P_1$ and $c_2$ has been used to generate the pattern $P_2$.

Applying $P_1$ on the page, we obtain the matches shown in the lower part of the figure. As can be seen, the values of the first and second match coincide with the examples $e_2$ and $e_3$, so the ranking for $c_1$ will be 2.

In turn, the match obtained by the pattern $P_2$ generated from $c_2$ do not coincide with any example, so the ranking for $c_2$ will be 0.
The pattern heuristic is similar in spirit to the sibling nodes heuristic, but it is slightly less effective, because the generated pattern may miss matches with examples that appear on the page but with a slightly different layout. For instance, in Fig. 13 the pattern generated from $c_1$ does not match with $e_4$, and $c_1$ gets a ranking of 2, whereas ideally it should be 3. This is not a major drawback in practice because we only need enough matches to discriminate between correct and wrong occurrences (one only match is enough in most cases). Anyway we only use the pattern heuristic as an alternative, when the sibling nodes heuristic is not applicable.
3.3.2 Proximity Heuristic

This heuristic builds on the observation that all the data values belonging to a certain valid example tend to be grouped near each other on the page. In conformance with this, the candidate example occurrences whose values are sparse on the page should be considered less promising. We can measure the “sparseness” of a candidate example occurrence $c_k$ by computing the sum of the distances between every two consecutive field occurrences belonging to $c_k$. The distance between two field occurrences is computed as the number of nodes between the last node of the first occurrence and the first node of the next occurrence (depth-first traversal order).

Another possibility for implementing this heuristic is actually computing the average “visual distance” on the page rendered by the browser between the fields of the candidate example occurrence. The visual distance between two DOM nodes may be computed by using the functionalities provided by current browser APIs. In our tests, the DOM distance measure performs almost as well as the visual distance “measure”. Thus, for convenience and efficiency reasons, our current implementation uses the DOM distance measure.

3.3.3 Detecting “fake” examples

As it has already been said, it may occur that a chosen example is not actually present on the page, although there are candidate field occurrences for all of their values (recall Fig. 6). That is, there exist candidate occurrences of the example, but all of them are wrong.

When the sibling nodes heuristic is applicable, this is dealt with in step 6 of the obtainBestAtomicCandidateOccurrences algorithm (Fig. 7), where examples that have a zero value in that heuristic for every candidate example occurrence are rejected.

When the sibling nodes heuristic is not applicable, we detect “fake” examples by analyzing overlaps on the page between the best candidate occurrences of several examples. The intuition
here is that if the best candidate occurrences of two examples overlap, then one of them is very probably wrong, since sources do not show the fields of a data element mixed with the fields of another. For instance, Fig. 14 shows two examples $e_1$ and $e_2$ and their best candidate occurrences $c_1$ and $c_2$ on a sample page. Since $c_2$ and $c_1$ overlap, the system detects that one of them must be wrong. The criterion we use is rejecting the example whose best candidate occurrence has a lower rank according to the proximity heuristic. For instance, in the situation shown in Fig. 14, we would reject $c_1$.

### 3.3.4 Dealing with too many candidate occurrences

Sometimes the number of candidate occurrences of an example is too high and it is not possible to compute the heuristic values for all of them. In these cases the algorithm needs to make a first preselection among all the possible combinations of the candidate field occurrences of the example to select in advance the most promising ones (step 4.a in Fig. 7).

The preselection process is based on the following steps. Given a certain number $N$, which is an upper limit to the number of candidate example occurrences we wish to generate, and an example $e=\{field_1=v_1, ..., field_r=v_r\}$ of type $T=\{field_1, ..., field_r\}$:
1. Find \( v_i \), the value of the field of \( e \) that has fewer occurrences on the page. The number of occurrences of \( v_i \) will be denoted \(|v_i|\).

2. Choose \( k \), the maximum number such as \( k^{r-1}\cdot|v_i| \leq N \). Since the total number of generated candidate example occurrences will be equal to the product of the number of candidate occurrences of each field, this will allow us to choose up to \( k \) occurrences of every other value \( v_{j,j \neq i} \) for each occurrence of \( v_i \), without surpassing the upper limit \( N \).

3. For each occurrence \( o \) of \( v_i \) on the page:
   a. For every \( v_{j,j \neq i} \) of \( e \), choose at most \( k \) occurrences of it on the page. If there are more than \( k \) occurrences of \( v_j \), the metrics used to select the \( k \) best ones are:
      i. The proximity with respect to \( o \) in the DOM tree using deep-first order.
      ii. The length of the common path in the DOM tree between the occurrence of \( v_j \) and \( o \). The longer the common path, the higher the occurrence of \( v_j \) is rated.

3.3.5 Dealing with Highly Time-Variable Data

When data that are to be extracted from a source vary at very short time intervals (e.g. stock quotes), we cannot assume any input example extracted from previous queries to remain completely valid in that source.

In these situations, the system distinguishes between the fields that are highly variable (e.g. the current change of a stock quote) and those we can assume will remain stable in at least some cases (e.g. the company acronym).

For “stable” fields we will compute the candidate field occurrences by using stored examples in the same way as in conventional sources.
For “highly variable” fields we will instead compute the candidate field occurrences by using a regular expression describing the format of the expected values (e.g. [0-9]+ “.” [0-9]+ for the current change of a stock quote). We also allow expected prefixes and suffixes to be included in the expression for the data values. Regular expressions are provided by the user during wrapper generation and they are stored by the maintenance system to be used whenever the source changes.

As we will remark in the experience section of the paper, these modifications make the techniques described in the previous sections maintain their effectiveness in this kind of sources.

3.4 Generalization to Types of Arbitrary Depth

In this section we show how to generalize the obtainBestAtomicCandidateOccurrences algorithm (Fig. 7) to examples of types of arbitrary depth. The main idea is based on applying recursively the algorithm to the tree of the example.

Let us consider the tree for an example with depth $k$. At the $(k-1)th$ level, all the non-atomic fields of the example are composed exclusively of atomic fields. Thus, they can be considered as if each of them were an example belonging to a type of depth 1.

For instance, in Fig. 15a two examples of type $T=\{TITLE, ARTIST, DATE, EDITION\{FORMAT, PRICE\}\}$ are shown. At the $(k-1)th$ level such data elements as $EX1.EDITION2=(format=“LP”, price=“18.61”)$ from Example 1, and $EX2.EDITION2=(format=“LP”, price=“17.34”)$ from Example 2 may be considered two unlabeled examples of type $EDITION=\{FORMAT, PRICE\}$. We will call these data elements “sub-examples of $(k-1)th$ level”.

Thus, as a first step we can consider every input example, obtain their sub-examples of $(k-1)th$ level and apply the obtainBestAtomicCandidateOccurrences algorithm to them to obtain their best candidate occurrences.
It is important to note that the `obtainBestAtomicCandidateOccurrences` algorithm may return more than one “best” candidate occurrence for each sub-example. For instance, in Fig. 15a the candidate occurrences marked as `co1` and `co2` will be both returned as “best” candidate occurrences for both sub-examples `EX1.EDITION1` and `EX2.EDITION1`.

Fig. 15. General algorithm operation
The following step is to substitute in $P$ the minimum HTML code containing each “best candidate occurrence” of the sub-examples of $(k-1)th$ level by a special unique markup on the page. For instance, in Fig. 15b the best occurrences of the sub-examples of type $EDITION$ have been substituted on the page for special markups of the form $SED\text{DITION}-I$, where $I$ is a number. If two sub-examples are formed by exactly the same data values and both are present on the page, then they will share at least two best candidate occurrences, which will be substituted on the page by the same special markup. For example, in Fig. 15b the occurrences $co_1$ and $co_2$ are both substituted for the special markup $SED\text{DITION}1$.

By performing this process for the $(k-1)th$-level sub-examples of all the input examples, the best candidate occurrences for non-atomic sub-elements of level $k-2$ can then be computed by simply invoking $obtainBestAtomicCandidateOccurrences$ with this representation of $P$ and using the special unique markups as values for the new atomic fields of $(k-2)th$ level (which have substituted the former non-atomic fields of $(k-1)th$ level).

For instance, in Fig. 15b the sub-examples of $(k-2)th$ level for, respectively, $EX1$ and $EX2$ are:


Note how atomic fields of the form $EDITION[i]$ have replaced the non-atomic field $EDITION$ in the examples.

It is important to note how the ambiguity we had at $(k-1)th$ level (where the candidate occurrences $co_1$ and $co_2$ were both considered as “best candidate occurrences” for both
EX1.EDITION1 and EX2.EDITION1) is solved at this level: the context provided by the other fields of the examples will allow the sibling nodes, pattern and proximity heuristics to determine co1 as the only “best candidate occurrence” for EX1.EDITION1 and co2 as the only “best candidate occurrence” for EX2.EDITION1.

By repeating recursively this process until the 0th level, we will obtain the best candidate occurrences for the complete examples. The algorithm is sketched in Fig. 16.

4 WRAPPER INDUCTION

In this section we overview our algorithm to reinduce the wrapper. In the section 4.1, we provide a quick introduction to DEXTL (which stands for Data Extraction Language), the wrapping language we use (see [12] for detail). In the section 4.2 we describe the induction process which
takes care of how the DEXTL programs are automatically generated from a set of labeled examples.

4.1 DEXTL Overview

A DEXTL program is composed of hierarchically structured DEXTL elements. Typically, a program for extracting data elements of a given type \( T \) will have a DEXTL element for each field from \( T \). The DEXTL elements will also be hierarchically arranged in the same way that the fields they represent are related in \( T \).

Thus, each DEXTL element can be either atomic (those without sub-elements) or non-atomic (those with sub-elements). Each non-atomic element has (among others not relevant here) the following parts: EXTRACTION clause (mandatory) and UNTIL clause (optional).

The EXTRACTION clause relates an element with its sub-elements and specifies how they are laid in the target source pages. This clause will be the basis to identify and extract the element data instances in the document. The UNTIL clause delimitates the end of the region of the document where the occurrences of the element must be searched for.

The basic structure used to specify the EXTRACTION and UNTIL clauses is called a DEXTL pattern. As we will see, a DEXTL pattern defines a sequence of data elements and separators among them.

The EXTRACTION clause will typically contain one DEXTL pattern for each possible visual layout of the element inside the target pages. The UNTIL clause uses DEXTL patterns to locate the end of the search region for the element. When the system is extracting instances of an element and an occurrence is found for a DEXTL pattern from its UNTIL clause, the system stops looking for occurrences of that element and continues searching for occurrences of its parent element (or finishes if there is not a parent element).
4.1.1 DEXTL Patterns

A DEXTL pattern is comprised of a list of text tokens which are laid consecutively in the document and which are delimited by tag-separators.

Text tokens represent text in the browser-displayed page. They are enclosed between '[' and ']' and they can be divided into portions by applying Perl5-like regular expressions. A placeholder (prefixed by the '$' character) can be assigned to the parts of the regular expression enclosed between '(' and ')'. The name may correspond either with an atomic field of the elements we wish to extract or with a special value called IRRELEVANT, which is used to represent non-constant strings appearing in the pattern, but which we do not wish to extract.

Tag-separators represent a regular expression concerning HTML document tags. For instance, we could define a tag-separator called EOL (EndOfLine) as follows: 

```
EOL = (<br>| </p>| </tr> (\n|\r|\t)* </tr>)
```

Though they can be defined to suit, DEXTL includes built-in tag-separators which are enough for the vast majority of situations concerning wrapper generation. Nevertheless, as we will see later, in the case of wrapper maintenance, tag-separators sometimes need to be dynamically built according to the characteristics of the pages we wish to deal with.

Each DEXTL pattern has a set of associated tag-separators, which are indicated through the SEPARATORS construction. All HTML tags that do not conform to a tag-separator are just ignored by the DEXTL extraction programs.

Example: Fig. 17 shows two search results from an electronic music shop. We wish to extract items having type ALBUM = {TITLE, ARTIST, DATE, EDITION{FORMAT, PRICE}}. Fig. 18 shows the respective fragment of HTML code, where tag attributes have been omitted. Fig. 19 shows a DEXTL program to extract the occurrences of the element ALBUM.
In the **EDITION** extraction pattern the only tag-separator used is *EOL*, while in the **ALBUM** extraction pattern the *TAB* ("\</td\>") and *ANCHOR* ("\</a\>") tag-separators are used.
4.2 Inducing DEXTL Patterns

Our induction algorithm will generate a set of DEXTL patterns for each non-atomic field of the type of the data elements we wish to extract. Then, it will compose a DEXTL program nesting the generated patterns according to the type structure. It will also find any UNTIL pattern needed.

Our basic algorithm for induction is `obtainAtomicPattern`, which is shown in Fig 20. This algorithm receives as inputs a type $T$ which is assumed to be exclusively composed of atomic fields, a set of examples for data elements of type $T$ and a page $P$ containing those examples (or, as usual, a set of pages considered as a unique page where their DOM trees are siblings). The algorithm outputs a set of DEXTL patterns for extracting occurrences of the desired elements of type $T$ from the target pages.
The \textit{obtainAtomicPattern} algorithm iterates over the examples to generate a candidate pattern based on each example. The process stops when the set of current patterns recognizes all the examples. Then, the set of results matched by each pattern is explored to remove redundant patterns (those which only match results also extracted by other patterns). The final step of the algorithm consists in lengthening the pattern to make it as less ambiguous as possible while still recognizing the same examples. This is made by adding tokens from $P$ at the end and the beginning of the candidate pattern.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example_diagram.png}
\caption{Generating candidate patterns}
\end{figure}
The following sub-sections describe in detail certain aspects of the algorithm: how candidate patterns are generated (4.2.1), how new tag separators are induced when needed (4.2.2) and how the algorithm is generalized to accept types of arbitrary depth.

4.2.1 Generating candidate patterns

The process for generating the candidate pattern for an example \( e_i \) is illustrated with an example in Fig. 21. The basic idea is as follows: first, we obtain an HTML portion containing the example and generate a preliminary DEXTL candidate pattern using the current set of tag-separators (we begin by using the former set of tag-separators used by the wrapper), and using the special placeholder \texttt{IRRELEVANT} to represent all the visible texts present in the HTML portion (Fig. 21 step 1).

Then, we execute the pattern on \( P \). The set of matches is probable to contain occurrences of other examples (the examples having the same “visual layout” than \( e_i \)). For instance in the Fig. 21, the preliminary pattern generated for the first example also matches with the third one.

Therefore, we identify these matches and use them to refine the pattern in the following way:

- We consider the text tokens that correspond with the field occurrences of the other input examples enclosed. We substitute the field occurrence by a placeholder with its name, and we find the longest common prefixes and suffixes of the strings surrounding the field occurrence. The common prefixes and suffixes will be considered as “fixed” in the refined regular expression for defining the text token (see Fig. 21 step 2 for an example).
The process of looking for common fixed prefixes and suffixes is also applied to the text portions which do not contain any field occurrence (using \textit{IRRELEVANT} as placeholder for the variable parts), but only if the number of other examples matched by the pattern exceeds a certain parameter $k$ (currently set to 5). This is a compromise to avoid generating “fixed” text sections that are too specific.

### 4.3 Inducing New Tag-Separators

The process gets more complicated if the current set of tag-separators is no longer valid when the source changes. This situation is detected in the step 3 of the algorithm for generating the candidate patterns. The system can correct this situation by applying the following rules:

1. The DEXTL interpreter assumes that the tag-separators cannot appear inside of the strings which match with the text tokens of the pattern (in other case, they would not actually be \textit{separators}). Therefore, if there are any occurrences of a tag-separator \textit{inside}
an occurrence of a field of an example, then that separator must be removed from the list of tag-separators. For instance, in the Fig. 22 (step 1) the tag separator EOL is not longer valid because it splits in two parts the commentary of the example news stories.

2) Any pair of text tokens (field names or IRRELEVANT) in a DEXTL pattern must be separated either by a text (specified as a prefix or suffix in the regular expression associated to the text token for the field) or by a tag-separator. If this does not happen for a preliminary DEXTL pattern, the system will add a new tag-separator by analyzing the HTML tags between both fields in all examples matching with the pattern, and choosing a common string of tags from either the beginning or the end (ignoring attributes), and ensuring that the chosen separator does not violate the former rule (it cannot appear inside of a field occurrence). For instance in Fig. 22 (step 2), after removing EOL as a valid tag-separator, the title and commentary of the news stories are not separated by neither a text nor a current tag-separator. Therefore, we need to generate a new tag-separator. “</A>” is a suitable election because it is a string of tags (of length 1) which appears between the title and commentary of the news stories but does not violate rule 1.

3) In the same way, the complete examples also need to be separated either by a text or by a tag-separator. If this does not happen for a preliminary DEXTL pattern, the system will add a new tag-separator in a similar way as in the previous case.

4.3.1 Generalization to types of arbitrary depth

The basic idea for the generalization of the obtainAtomicPattern algorithm to types of arbitrary depth is reminiscent of the one used to generalize the algorithms for labeling examples in section 3.4. We recursively build candidate patterns for all the non-atomic sub-elements of the type and assemble them hierarchically to make up a DEXTL program. An additional difficulty for the
generalized algorithm is computing the *UNTIL* clauses that the DEXTL interpreter needs in order to detect the end of the region where the occurrences for a certain sub-element are found. This is overcome by searching a DEXTL pattern with zero matches inside the examples of the sub-element and which appears always after the last example of the sub-element.

5 EXPERIMENTS

To evaluate the effectiveness of our approach we monitored a set of Web sites during six months. We selected sources from different domains presenting different characteristics and data types. For instance, we included certain sources with some highly variable data fields (e.g. *Nasdaq*) to test our techniques in that scenario. We also included some sources where the contents change very frequently (e.g. *New York Times*).

For every Web site, we first generated a wrapper and used it on a daily basis to execute different queries. The system automatically stored some of the queries and their results according to the techniques explained in section 2. When a change in the format of the pages of a Web site was detected, the system generated a new set of labeled examples and used them as input to induce a new wrapper for the site (in these experiments we used our own wrapper induction algorithm for this stage). Once the wrapper had been regenerated, we tested it with a new set of pages from the source obtained through different queries.

We will quantify the results at two different points of the regeneration process: after generating the new set of labeled examples and after the reinduction stage. At the first point, we can measure the effectiveness of our techniques for generating new correct labeled examples. At the second point, the wrapper is completely regenerated and we can measure the fitness of the generated examples to re induce the wrapper using our induction algorithm.
To quantify the results of the stage of generating a new set of labeled examples we define the following metrics:

- LN: number of input examples (from the stored queries).
- LPE: number of input examples that are still present in the current HTML pages of the source after the change.
- LTE: number of total examples labeled by the system.
- LCE: number of examples correctly labeled by the system.
- Labeling Recall (LR) = LCE/LPE. It represents the ratio between the correctly labeled examples and all the examples that should be labeled on the pages.
- Labeling Precision (LP) = LCE/LTE. It represents the ratio between the correctly labeled examples and the total number of labeled examples.

To quantify the results once the wrappers have been regenerated we define the following metrics:

- N: number of data items that should be extracted from the test pages.
- TE: number of total data items extracted by the regenerated wrapper from the test pages.
- CE: number of correct data items extracted by the regenerated wrapper from the test pages.
- Recall (R) = CE / N. It represents the ratio between the correctly extracted data items and all the data items that should be extracted from the test pages.
- Precision (P) = CE / TE. It represents the ratio between the correctly extracted data items and all the data items that have been actually extracted from the test pages.

5.1 Analysis of Results

Table 1 lists the subset of the monitored sources where any changes occurred during the period of study. Each of the sources underwent just one format change except for Barnes & Noble, Amazon Magazine, Espacenet and Yahoo People, where two changes occurred (in these cases we
computed the averages of the obtained results). The last column indicates if the sibling nodes heuristic was applicable in the source after the change.

Some of the changes were relatively small. For instance, in Yahoo People a new column was added to the table that renders the people’s information. On the other hand, some sources changed their underlying templates completely (for instance, Barnes & Noble).

Table 2 shows the values of the metrics computed for the labeling process. A first conclusion is that the ratio of preserved examples ($LPE\%$) is high for almost all the monitored sources. This supports the idea of using results from previous queries as a good basis on which to generate a new set of examples for reinduction.

The only source where the ratio of preserved examples is very low is New York Times, in which case only 19% of the examples are preserved. Nevertheless, as we will see, the system showed a high effectiveness even in this case.

### Table 1

**List of Sources and Data Fields Extracted from Them**

<table>
<thead>
<tr>
<th>Source</th>
<th>Data fields</th>
<th>Sibling nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllBooks4Less (Book)</td>
<td>title, author, format, price, quantity</td>
<td>No</td>
</tr>
<tr>
<td>Amazon (Book)</td>
<td>title, author, format, price, availability</td>
<td>Yes</td>
</tr>
<tr>
<td>Amazon (Magazine)</td>
<td>title, price, unit price, issues, months</td>
<td>Yes</td>
</tr>
<tr>
<td>Barnes&amp;Noble (Book)</td>
<td>title, author, format, price, availability</td>
<td>Yes</td>
</tr>
<tr>
<td>BigBook (Person)</td>
<td>name, street address, city, state, phone</td>
<td>Yes</td>
</tr>
<tr>
<td>Cordis (Project)</td>
<td>title, date, relevance, status, program acronym, project acronym, reference, start date, end date, abstract</td>
<td>Yes</td>
</tr>
<tr>
<td>Delphion (Patent)</td>
<td>title, publication, date, store</td>
<td>Yes</td>
</tr>
<tr>
<td>Discoweb (Album)</td>
<td>title, artist, date, edition: {format, price}</td>
<td>Yes</td>
</tr>
<tr>
<td>Espacenet (Patent)</td>
<td>title, publication, date, inventor, applicant, IPC</td>
<td>No</td>
</tr>
<tr>
<td>Nasdaq (Flash Quote)</td>
<td>symbol, last sale, share volume, market</td>
<td>Yes</td>
</tr>
<tr>
<td>NewYorkTimes (News Story)</td>
<td>title, author, summary</td>
<td>Yes</td>
</tr>
<tr>
<td>YahooPeople (Person)</td>
<td>name, street address, city, state, phone</td>
<td>Yes</td>
</tr>
<tr>
<td>YahooQuotes (Quote)</td>
<td>symbol, time, trade, change, % change, volume</td>
<td>Yes</td>
</tr>
</tbody>
</table>
In the sources with data fields considered as highly variable (Nasdaq, Yahoo Quotes, AllBooks4Less), we computed the percentage of preserved examples without considering those fields. As can be seen, the ratio is very high, which indicates that the values of the remaining fields in these sources are very stable.

The LR and LP metrics give a 100% value for all the sources except for AllBooks4Less, where the system did not discard two “fake” examples. The errors were caused because the sibling nodes heuristic was not applicable and there were some rare cases where the proximity heuristic failed.

Table 3 shows the results calculated after complete wrapper regeneration. The second and third columns show the values computed for the Recall and Precision metrics. The fourth column indicates the number of items that should be extracted from the pages used to test the wrapper. The fifth column shows the number of input examples to the reinduction algorithm (i.e. the examples labeled in the previous stage). Remember that all examples are correctly labeled in all sources except in AllBooks4Less where 2 of the 64 examples are erroneous. The sixth column shows the number of extraction patterns created by our induction algorithm for each wrapper. As
we have already seen in section 2, an extraction pattern can be roughly matched to a different “visual representation” of the data items in the source.

The basic factor influencing the results at this stage is that the induction process benefits from having at least a certain number of examples for each possible visual representation, so the appropriate extraction rules can be induced.

So recalling the regenerated wrapper depends on the suitability of the set of labeled examples to cover the whole range of those different visual representations. If the examples do not cover this entire range, then, even when 100% of the examples are correctly labeled at the previous stage, the system will still not reach 100% for the recall and precision metrics. Consequently, the worst results were obtained in Amazon Book, which is, by far, the source with the largest variety of visual representations, and in New York Times, which is the source with more data variability (only 19% of the stored examples are preserved).

TABLE 3
WRAPPER REGENERATION METRICS

<table>
<thead>
<tr>
<th>Source</th>
<th>R (%)</th>
<th>P (%)</th>
<th>N</th>
<th>LTE</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllBooks4Less (Book)</td>
<td>100</td>
<td>97.9</td>
<td>1000</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>Amazon (Book)</td>
<td>86.8</td>
<td>93.4</td>
<td>1000</td>
<td>147</td>
<td>27</td>
</tr>
<tr>
<td>Amazon (Magazine)</td>
<td>99.8</td>
<td>100</td>
<td>1000</td>
<td>50</td>
<td>7</td>
</tr>
<tr>
<td>Barnes&amp;Noble (Book)</td>
<td>98.2</td>
<td>100</td>
<td>1000</td>
<td>65</td>
<td>8</td>
</tr>
<tr>
<td>BigBook (Person)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>Cordis (Project)</td>
<td>97.8</td>
<td>100</td>
<td>1000</td>
<td>99</td>
<td>5</td>
</tr>
<tr>
<td>Delphion (Patent)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>Discoweb (Album)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>87</td>
<td>2 / 1 (1)</td>
</tr>
<tr>
<td>Espacenet (Patent)</td>
<td>99.5</td>
<td>100</td>
<td>1000</td>
<td>196</td>
<td>12</td>
</tr>
<tr>
<td>Nasdaq (Flash Quotes)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>NewYorkTimes (News Story)</td>
<td>94.5</td>
<td>100</td>
<td>200</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>YahooPeople (Person)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>YahooQuotes (Quote)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
</tbody>
</table>

(1) Album type / Edition subtype
6 RELATED WORK

Semi-automatic wrapper generation has been an active research field for years [4,6,7,12] (see [8] for a survey). These works are only concerned with wrapper generation and do not support automatic wrapper maintenance.

[5,9] address the wrapper verification problem and, thus, they are complementary to our work. [9,10,11] have addressed the problem of wrapper maintenance. [9] uses information collected during previous wrapper operation to perform an automatic process for labeling a set of new examples, thus producing the input for their induction algorithm. The process for identifying the correct candidate field occurrences does not take into account neither the context provided by the candidate occurrences of the other fields forming the element nor by the other examples of the same type. In addition, they do not deal with the maintenance of wrappers for pages containing “lists” of data elements (they only consider sources where each page contains information of only one data item). In our experience, the need for extracting information from list pages is very common.

[10] identifies candidate field occurrences by assuming some structural features to remain invariant when sources change. These features are annotations (text strings that identify a certain data field such as “Title” or “Price”), hyperlink features and syntactic features (it assumes that the values will conform to a given regular expression). Then they use the identified candidate field occurrences and the schema of the target data elements to identify a set of blocks in the page that conforms to the structure and content suggested by the data schema. In sources where the assumption of invariance of annotations and hyperlinks does not hold, the system will fail to identify correct examples.
[11] proposes a wrapper reinduction approach, which iteratively uses the extracted data at a given time \( t \) to regenerate the wrapper at a later moment \( t+s \), where \( s \) is small. Their approach can only induce LR-wrappers [6], which are not expressive enough to deal with most modern Web sites. In addition, they do not deal with the problem of correctly labeling the examples in the target pages after a source change.

Another difference with respect to [9,10,11] is that our system takes into account the context provided by other examples in order to choose the best candidate occurrences for an example. This improves the accuracy of the labeling process and makes the approach less vulnerable, when the target pages have several portions that conform to the underlying schema.

[15] proposes an ontology-based schema for wrapper generation. In this schema, an expert creates an ontology describing a particular domain of interest, and the wrappers for sources compliant with the domain description can be automatically generated by the system. In this approach the wrappers are resilient to changes, as long as the ontology continues describing the source after the change. Defining such ontologies for non-trivial domains is a very challenging task, since the differences between sources in the same domain can be vast.

The works presented in [1] and [3] deal with automatic wrapper generation and are also related to our system to some extent. They take as input a set of example pages of the same class and automatically induce the underlying template they conform to. [14] uses a similar approach but they do not need several pages; they only need a single page containing several structured records following a similar layout. These approaches have the advantage of not requiring input examples for wrapper generation (at the cost of having less information on which to base the building of accurate wrappers), but they do not annotate the extracted results. While human post-annotation of the data is acceptable in wrapper generation, it is not so for automatic wrapper
maintenance. In [2] an approach is presented for automatic annotation of the extracted data using the texts surrounding the extracted items on the page. Nevertheless, it cannot deal with the common situations, where some data fields have not associated labels on the page.

7 CONCLUSIONS

In this paper we have presented techniques for the generation and automatic maintenance of wrappers for semi-structured Web sources. Our approach is based on using inductive learning algorithms for generating wrappers. For automatic maintenance, our system collects some query results during wrapper operation and, when the source changes, they are used to generate a new set of labeled examples which constitute the input to induce the new wrapper.

The main step involved in this process is correctly labeling on the new pages the examples collected during previous wrapper operation. Our approach is based on three heuristics ("sibling nodes", "pattern" and "proximity"), which arise from observations of the typical ways to arrange semi-structured information on a page or set of pages. We have experimentally tested our techniques for a variety of real-world Web sources, obtaining a high degree of effectiveness.

8 REFERENCES


