HW-STALKER: A Machine Learning-based System for Transforming QURE-Pagelets to XML

Vladimir Kovalev [a], Sourav S Bhowmick [a] and Sanjay Madria [b]

[a] *School of Computer Engineering, Division of Information Systems, Nanyang Technological University, Singapore 639798*

[b] *Department of Computer Science, University of Missouri-Rolla, Rolla 65409*

Abstract

In this paper, we address the problem of extracting and transforming dynamically generated hyperlinked hidden web query results to XML. Our approach is based on the STALKER approach. As STALKER was designed to extract data from a single web page, it cannot handle a set of hyperlinked pages. We propose an algorithm called *HW-Transform* for transforming hidden web query results (also called *QURE-Pagelets*) to XML format using machine learning by extending STALKER to handle hyperlinked hidden web pages. One of the key features of our approach is that we identify and transform *key attributes* of query results into XML attributes. These *key attributes* facilitate applications such as change detection and data integration by efficiently identifying *related* or *identical* results. Based on the proposed algorithm, we have implemented a prototype system called HW-STALKER using Java. Our experiments demonstrate that *HW-Transform* shows acceptable performance for transforming QURE-pagelets to XML.

Key words: Hidden Web, dynamic content, identifiers, facilitators, STALKER, XML, QURE-Pagelets.

1 Introduction

Current-day web crawlers retrieve content only from a portion of the Web, called the *publicly indexable Web* (PIW) [16]. This refers to the set of web pages reachable exclusively by following hypertext links, ignoring search forms and pages required authorization or registration. However, recent studies [17,10] observed that a significant fraction of Web content lies outside the PIW. A

Email addresses: assourav@ntu.edu.sg, madrias@umr.edu (Sanjay Madria).

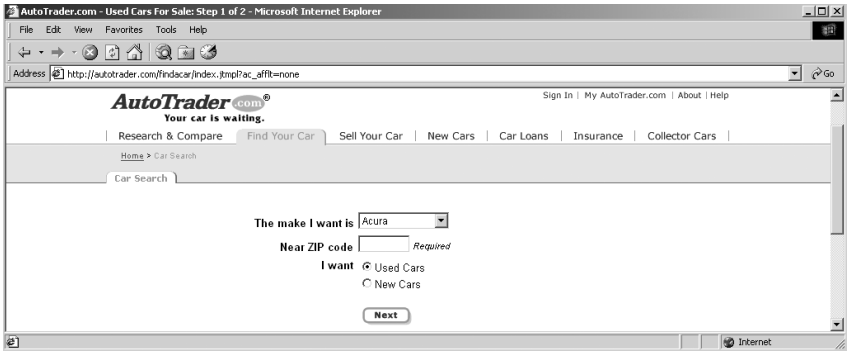
great portion of the Web is "hidden" in databases and can only be accessed by posing queries over these databases using search forms (lots of databases are available only through HTML forms)[10]. This portion of the Web is known as the *hidden Web* or the *deep Web* [10]. Pages in the hidden Web are dynamically generated in response to queries submitted via the search forms. We illustrate such hidden web queries with an example.

Example 1 `AutoTrader.com` (at `http://www.autotrader.com`) is the largest used car web site with over 2 million new and used vehicles listed for sale by private owners, dealers, and manufacturers. To get any information on the listed vehicles, a user should first specify search conditions. Figure 1 depicts the search interface available on the `AutoTrader.com` site. This interface is composed from two consecutive pages. In the first page, a user specifies the car make he is searching for. In the second page, a user specifies the details of the car such as model, year, price range, color, etc. After submitting the search query, a user gets a list of cars relevant to this query. Figure 2 represents a set of pages returned as the result of searching for *Ford* cars at `AutoTrader.com` on 02 July, 2003. There are 500 car descriptions returned to a user. The first page contains short descriptions of 25 cars. Each such short description provides a link to a separate page containing more details on the particular car. There is also a link to a page containing the next 25 car descriptions that is formatted in a similar way as the first page. The second page is linked to the third page, and so on. Note that all the pages in this set are generated dynamically. This means that every time a user queries `AutoTrader.com` with the same query, all the resulting pages and the links that connect them to one another are generated from the hidden web database anew. As the results are always ordered by some criteria (e.g. price, year, mileage, year, etc.), the description of the same car may appear in different positions each time a particular query is executed. ∎

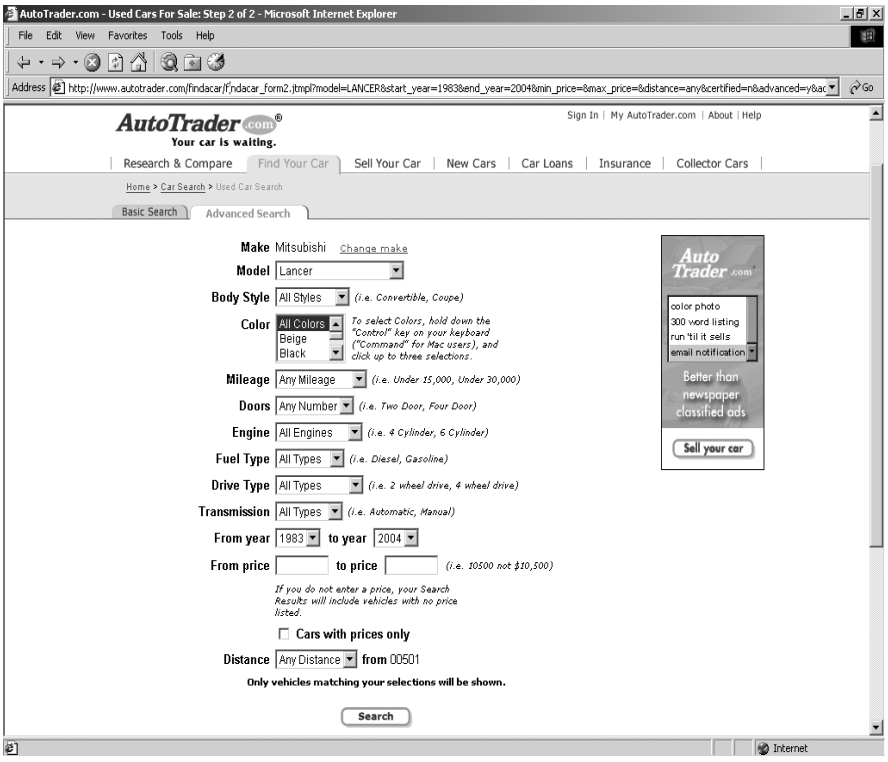
The task of harvesting information from the hidden web can be divided into following four steps.

(1) Formulate a query or search task description;
(2) Discover sources that pertain to the task;
(3) For each potentially useful source, fill in the source's search form and execute the query;
(4) Extract query results from result pages as useful data is embedded into the HTML code.

In this paper, we will assume that the task is formulated clearly (Step 1). Step 2, *source discovery*, usually begins with a keyword search on one of the search engines or a query to one of the web directory services. The works in [4,18,6] address the resource discovery problem and describe the design of topic-specific PIW crawlers. Techniques for automatically querying hidden



(a) Step 1 of 2.



(b) Step 2 of 2.

Fig. 1. `AutoTrader.com`: Search interface.

web search forms (Step 3) has been proposed in [12,21,19,9]. These techniques allow a user to specify complex queries to hidden web sites that are executed as combination of real queries. In this paper, we focus on Step 4.

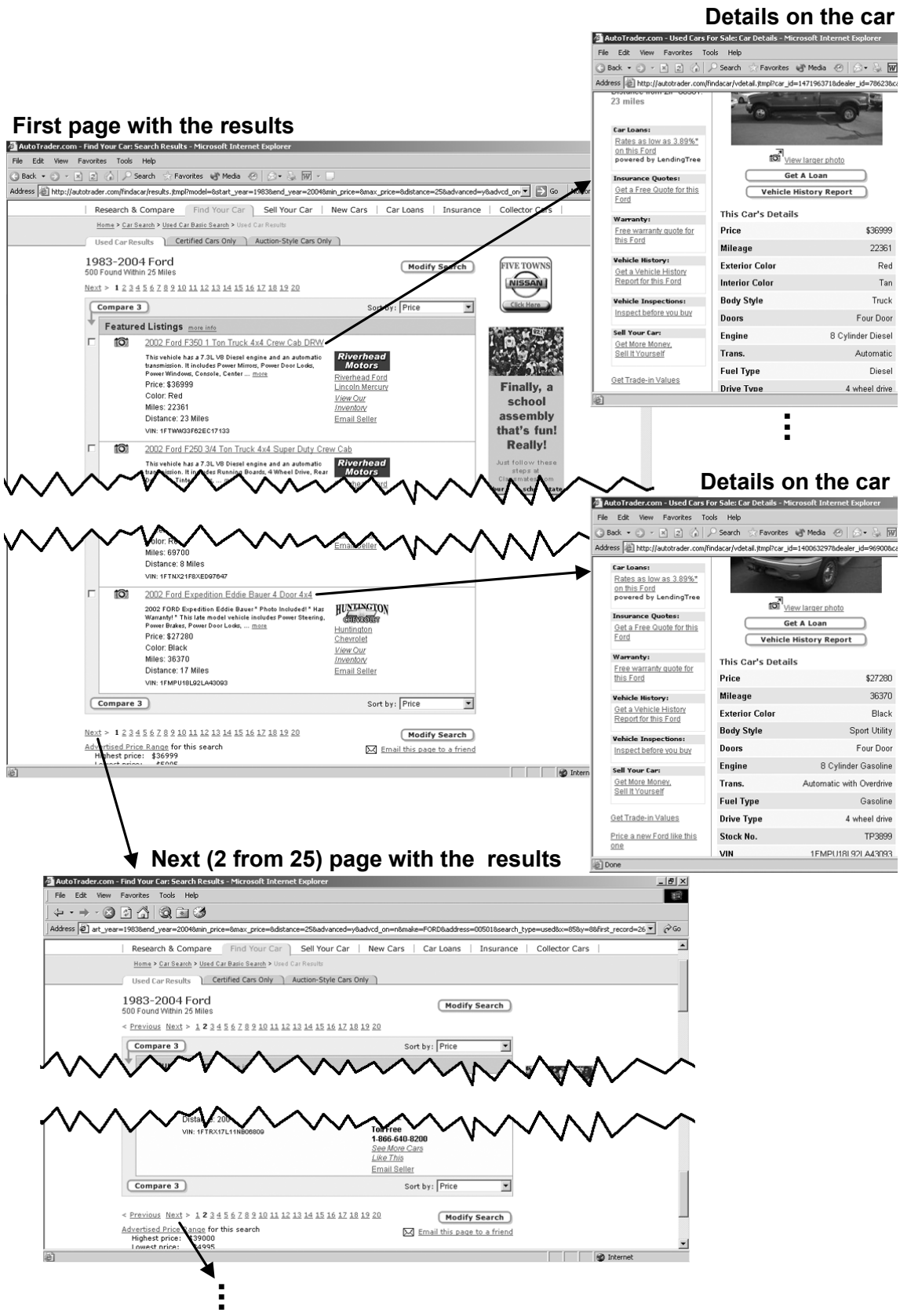


Fig. 2. A set of pages returned as the result of searching for *Ford* on 2 July, 2003.

1.1 Motivation

Dynamically generated web pages typically consist of a handful of presentation region types. Three common examples include [3]:

- The *query-result regions*, which present the primary content directly related to a query posed on the search interface of the content provider. Some web sites support multiple primary content regions.
- The *advertisement region*, which presents the information about other products offered by the content provider or about related products offered by other companies.
- The *navigational region*, which presents a collection of navigational links, often to other web sites provided by the same content provider.

In this paper, we focus on the transformation of data in query-result regions to XML. We introduce the concept of ***QU****ery-****RE****sult Pagelet*[1] (*QURE-Pagelet* for short) to refer to the query results related content region in a dynamic page generated by the execution of a query on a hidden web site. Extracting relevant results automatically from QURE-pagelets is a challenging problem. First, the search and the extraction of the required data from the dynamic pages are highly complicated tasks as each web form interface is designed for human consumption and, hence, has its own method of formatting and layout of elements on the page. For instance, Figure 2 depicts the original `AutoTrader` result page with formatting and non-informative elements (such as banners, advertisements, etc.). Accordingly, extraction tools must be able to filter out the relevant QURE-Pagelets from the pages. Second, is the structural complexity of hidden web query results. The search query usually returns not a single HTML page, but a set of HTML pages. Most hidden web sites use hyperlinks to connect these HTML pages. However, some sites use client-side scripts like JavaScript for this purposes. These scripts are used to generate hyperlinks on-demand based on parameters provided by user on submission. Unfortunately, it is computationally hard to automatically analyze client-side scripts.

We present HW-STALKER, a prototype system for extracting relevant QURE-pagelets and transforming them to XML using machine learning technique. Our motivation to transform QURE-pagelets to XML is the following. Hidden web data is HTML-formatted and every hidden web site generates it in its own fashion. Thus it becomes extremely difficult and cumbersome to develop generalized techniques that can be used for hidden web data integration, change detection to hidden web data [13], warehousing hidden web data etc. Consequently, it is important to develop a technique for transforming hidden

[1] The term *pagelet* was first introduced in [1] to describe a region of a web page that is distinct in terms of its subject matter or its content

web data to more structured format (eg. XML) so that we can develop such generalized techniques for hidden web data. A shorter version of this paper appeared in [14].

1.2 Overview

We propose an algorithm called *HW-Transform* for transforming QURE-pagelets to XML format. Our approach is based on the STALKER technique [11,20]. We use STALKER because apart from being easy to use, in most cases it needs only couple of examples to learn extraction rules, even for documents containing lists. The extraction rules are typically very small, and consequently, they are easy to induce. This is a crucial feature because from the user's perspective it makes the wrapper induction process both fast and painless. Moreover, STALKER models a page as unordered tree and many hidden web query results are unordered. However, STALKER was designed to extract data from a single web page and cannot handle a set of hyperlinked pages. Hence, we need to extend the STALKER technique to extract results from a set of dynamically generated hyperlinked web pages.

We use machine learning-based technique to induce the rules for this transformation. The process of transforming QURE-pagelets from HTML to XML can be divided into three steps:

(1) Constructing *extended* $\mathcal{EC}$ description [11,20] describing the hidden web query results. In contrast to the STALKER approach, on of the key feature of our approach is that a user maps special *key attributes* (*identifiers* and *facilitators*) of query results into XML attributes. These *key attributes* facilitate change detection, data integration etc. by efficiently identifying *related* or *identical* results. We shall elaborate on the importance of these key attributes in the context of change detection to hidden web data in Section 3.
(2) Learn extraction rules based on examples labeled by the user. A GUI allows a user to mark up several pages on a site, and the system then generates a set of extraction rules that accurately extract the required information. We do not discuss this step here as it is similar to that of STALKER approach.
(3) Transforming QURE-pagelets from HTML to XML using extended $\mathcal{EC}$ tree with assigned rules. We discuss this step in Section 4. Note that we do not address transformation related to client-side scripts in this paper.

The rest of the paper is organized as follows. Section 2 discuss briefly the STALKER approach. In Section 3, we propose how to adapt the STALKER approach to model and transform hyperlinked QURE-pagelets to XML. Section 4 presents *HW-Transform*, a formal algorithm for transforming QURE-pagelets to XML format. We discuss the implementation of HW-STALKER and highlight

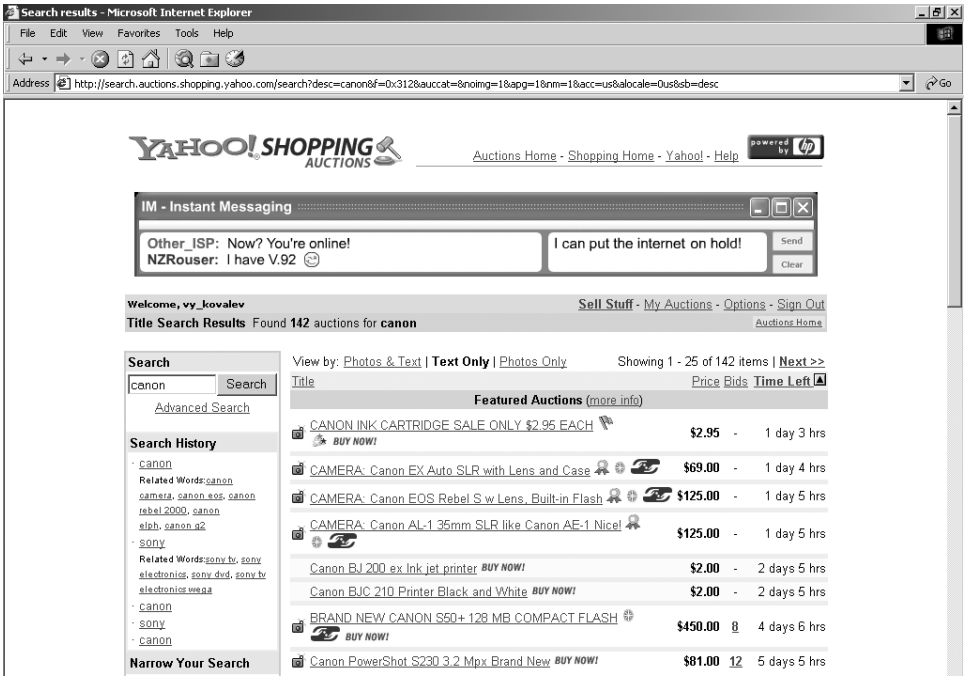


Fig. 3. Search results for *canon*.

some experimental results in Section 5. Section 6 discusses related research in this area. Finally, the last section concludes the paper.

2 STALKER Approach

In this section, we present the STALKER approach for wrapper construction that enables users to turn web pages into relational information sources [11,20]. STALKER is a machine learning based approach where *Embedded Catalog*($\mathcal{EC}$) formalism is used to describe the content of a web page. The $\mathcal{EC}$ description of a page is a tree-like structure (also called $\mathcal{EC}$ *tree*) in which the leaves represent the relevant data. The internal nodes (elements) of the $\mathcal{EC}$ tree represent *lists* of k-tuples, where each item in the k-tuple can be either a leaf l or another list L (in which case L is called an embedded list). For example, Figure 4 displays the sample $\mathcal{EC}$ description of `Yahoo!Auctions` (Figure 3). At the top level this page contains 3 tuples: a total number of auctions denoted by leaf element *Total*, a search keyword denoted by leaf element *Keyword*, and an embedded list of *Auction*s, respectively.

2.1 Extracting Data from a Document

In order to extract the items of interest, a wrapper uses the $\mathcal{EC}$ description of the document and a set of extraction rules. For each node in the tree, the wrapper needs a rule that extracts that particular node from its parent.

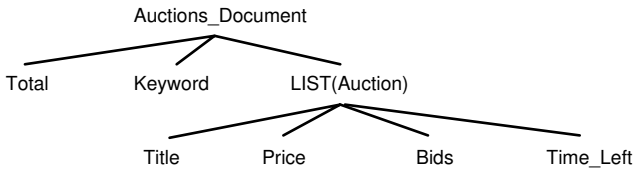


Fig. 4. $\mathcal{EC}$ description of `Yahoo!Auctions` list of auctions from Figure 3.

E1: Canon BJ 200 ex Ink jet printer $2.00- 2 days 5 hrs

E2: Canon BJC 210 Printer Black and White $2.00- 2 days 5 hrs

E3: BRAND NEW CANON S50+ 128 MB FLASH $450.008 4 days 6 hrs

E4: Canon PowerShot S230 3.2 Mpx Brand New $81.0012 5 days 5 hrs

Fig. 5. Four examples of auction descriptions.

Additionally, for each list node, the wrapper requires a *list iteration rule* that decomposes the list into individual tuples. Given the $\mathcal{EC}$ tree and the rules, item can be extracted by simply determining the path P from the root to the corresponding node and by successively extracting each node in P from its parent. If a parent of a node x is a list, the wrapper first applies the list iteration rule and then applies the x's extraction rule to each extracted tuple.

In the STALKER framework, a document is treated as a sequence of tokens S (e.g. words, HTML tags, wildcards, etc.). The content of the root node in $\mathcal{EC}$ tree is the whole sequence S, while the content of each of its children is a subsequence of S respectively. Generally, the content of an arbitrary node represents a subsequence of the content of its parent.

A key idea of STALKER approach is that the extraction rules are based on "landmarks" that enable a wrapper to locate the start and end of the item within the page. For example, let us consider the four examples of auction descriptions in Figure 5. In order to identify the beginning of the price, we can use the rule

$$\mathbf{R1} = \mathit{SkipTo}(\texttt{<b>})$$

which has the following meaning: start from the beginning of the document and skip everything until you find the `<b>` landmark. Note that **R1** is applied to the content of the node's parent, which in this particular case is the auction list. **R1** is called a *start rule* because it identifies the beginning of the price. One can write a similar *end rule* that finds the end of the price *from the end* of the document towards its beginning. These rules are not unique. That is, **R1** is by no means the only way to identify the beginning of the price.

To deal with variations in the format of the documents, the STALKER extraction rules allow the use of disjunctions. For example, let us assume that the auctions that do not have any bid appears in italic (see E1 and E2 in Figure 5), while the other ones are displayed as bold (eg. E3, E4). STALKER can extract all the bids based on the disjunctive start rule

either *SkipTo*(<i>)
or *SkipTo*()

Disjunctive rules are ordered lists of individual disjuncts. Applying a disjunctive rule is a straightforward process in STALKER: the wrapper successively applies each disjunct in the list until it finds the first one that matches.

2.2 Learning Extraction Rules

The authors have developed a *hierarchical* wrapper induction algorithm that learns extraction rules based on examples labeled by the user. A GUI allows a user to mark up several pages on a site, and the system then generates a set of extraction rules that accurately extract the required information. Specifically, the input to stalker algorithm consists of sequences of token representing the prefixes that must be consumed by the induced rule. To create such training examples, the user has to select a few sample pages and use GUI to mark up the relevant data (i.e., the leaves of the $\mathcal{EC}$ tree). Once a page is marked up, the GUI generates the sequences of tokens that represent the content of the parent p, together with the index of the token that represents the start of item x and uniquely identifies the prefix to be consumed.

STALKER exploits the hierarchical structure of the source to constrain the learning problem. For instance, instead of using one complex rule that extracts all auctions, titles, bids, price, etc. from a page, STALKER takes a hierarchical approach. First it applies a rule that extracts the whole list of auctions; then it use another rule to break the list into tuples that correspond to individual auctions; finally, from each such tuple the algorithm extracts the title, price, bids, and time left of the corresponding auction.

To generate a rule that extracts an item x from its parent p, STALKER takes a list of pairs $(T_i,\ Idx_i)$ as input, where each sequence of tokens T_i is the content of an instance of p, and $T_i[Idx_1]$ is the token that represents the start of x within p. Any sequence $S ::= T_i[l], T_i[2], \ldots, T_i[Idx_i - 1]$ (i.e., any instance of prefix of p with respect to x) represents a positive example, while any other sub-sequence or super-sequence of S represents a negative example. STALKER, tries to generate a rule that accepts all positive examples and rejects all negative ones.

STALKER is a *sequential covering* algorithm that, given the training examples

E, tries to learn a minimal number of *perfect disjuncts* that cover all examples in E. By definition, a perfect disjunct is a rule that covers at least one training example and on any example the rule matches it produces the correct result. STALKER first creates an initial set of candidate-rules C and then repeatedly applies the following three steps until it generates a perfect disjunct:

- select most promising candidate from C
- refine the candidate
- add the resulting refinements to C

Once STALKER obtains a perfect disjunct P, it removes from E all examples on which P is correct, and the whole process is repeated until there are no more training examples in E. STALKER uses two types of refinements: *landmark refinements* and *topology refinements*. The former makes the rule more specific by adding a token to one of the existing landmarks, while the latter adds a new 1-token landmark to the rule. More details on this algorithm can be found in [20].

Example 2 Let us consider the auction list from Figure 3. Figure 5 depicts four examples (*E1*, *E2*, *E3*, *E4*) of auction descriptions matching *Auction* node in the $\mathcal{EC}$ tree in Figure 4. We want to learn a start rule for the bids. STALKER proceeds as follows. First, it selects an example, say $E1$, to guide the search. Second, it generates a set of initial candidates, which are rules that consist of a single 1-token landmark; these landmarks are chosen so that they match the token that immediately precedes the beginning of the bids in the guiding example. The last token to be consumed in *E1* is "<i>". *HtmlTag* and *Anything* are the wildcards that match this token. Consequently, STALKER creates three initial candidates:

- **R1**=*SkipTo*(<**i**>)
- **R2**=*SkipTo*(**HtmlTag**) (stops as soon as it encounters an HTML tag)
- **R3**=*SkipTo*(**Anything**)

As **R1** is a perfect disjunct, STALKER returns **Rl** and the first iteration ends. During the second iteration, the algorithm is invoked with the uncovered training examples *E3* and *E4*. After this step, it returns rule **R4**=*SkipTo*(</**b**><**b**>) covering both examples. Consequently, STALKER stops the learning process and returns the disjunctive rule **either R1 or R4**. ∎

2.3 Summary

In summary, STALKER has the ability to wrap a large variety of sources. The experimental results described in [20] show that in most cases STALKER needs only couple of examples to learn extraction rules, even for documents containing lists. The number of required examples is small because the $\mathcal{EC}$ description

(a) Chain links on `AutoTrader.com`.

(b) Chain links on `ArchitectureWeek.com`.

(c) Chain links on `CiteSeer.com`.

(d) Chain links on `IMDb.com`.

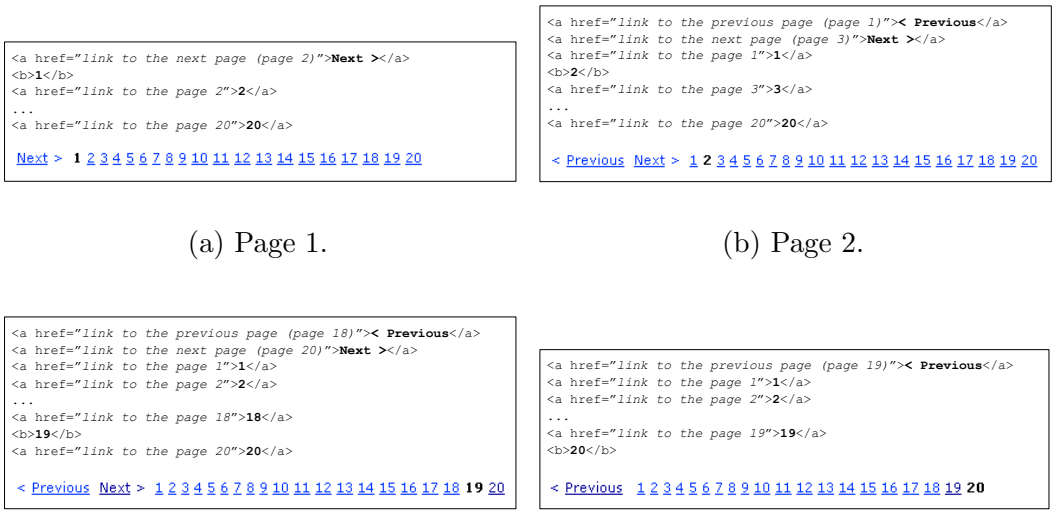
Fig. 6. Chains links on different hidden web sites.

of a page simplifies the problem tremendously: as the web pages are intended to be human readable, the $\mathcal{EC}$ structure is generally reflected by actual landmarks on the page. STALKER merely has to find the landmarks, which are generally in the close proximity of the items to be extracted. In other words, the extraction rules are typically very small, and consequently, they are easy to induce. This is a crucial feature because from the user's perspective it makes the wrapper induction process both fast and painless.

3 Transforming QURE-Pagelets

In this section, we present our approach to transform the hidden web query results or QURE-pagelets to XML format. Our approach is based on the STALKER technique [11,20]. Recall that we use the STALKER because apart from being easy to use, in most cases it needs only couple of examples to learn the extraction rules, even for documents containing lists. Moreover, the STALKER models a page as unordered tree and many hidden web query results are unordered. However, the STALKER was designed to extract data from a single web page and cannot handle a set of hyperlinked pages. Hence, we need to extend the STALKER technique to extract results from a set of dynamically generated hyperlinked web pages.

As mentioned in Section 1, the process of transforming the hidden web query results to XML can be divided into three steps: (1) Constructing the *extended* $\mathcal{EC}$ tree [11,20] describing the hidden web query results. (2) Learning extraction rules to precisely locate relevant information from a page by providing learning examples. We do not discuss this step here as it is similar to that



(a) Page 1. (b) Page 2.

(c) Page 19. (d) Page 20.

Fig. 7. `AutoTrader.com`: chain links in query results.

of the STALKER approach. (3) Transforming the results from HTML to XML using extended $\mathcal{EC}$ tree with assigned rules. We now elaborate on Steps (1) and (3).

3.1 Modelling QURE-Pagelets

As the hidden web results are distributed between a set of pages, we need a general model to represent these pages. In other words, we should model the links between a collection of hyperlinked hidden web pages. We distinguish these links into two types - the *chain links* and the *fanout links*. When the pages returned by a query are linked to one another by a set of links, we say that the links between them are the *chain links*. Examples of chain links in results from four different sites are depicted in Figure 6. When the result returned by a query contains hyperlinks to pages with additional information, we say that these links are the *fanout links*. For example, consider the Figure 2. As there are 500 result matches, these results are distributed in 20 pages. These pages are connected by *chain links* - each page contains links to the next (except the last page) and to the previous page (except the first page) in the set. However, these pages only contain short summary of each result. The full details of each result can be found in an additional page by clicking on the hyperlink in each result (e.g., the link labeled "2002 Ford Expedition Eddie Bauer 4 Door 4 × 4" in Figure 2). This link is called the *fanout link*.

The STALKER uses *Embedded Catalog*($\mathcal{EC}$) formalism to model an HTML page. This formalism is used to compose tree-like structure of the page based on *List* and *Element* nodes. The $\mathcal{EC}$ tree nodes are unordered. Thus, to be able to apply the STALKER to a set of pages we should extend the $\mathcal{EC}$ formalism for modelling a set of pages. We add three new types of nodes (*chain*, *fanout*, and *semantic* nodes) to the $\mathcal{EC}$ tree formalism. The new formalism is called the *Hidden Web Embedded Catalog* ($\mathcal{HW}-\mathcal{EC}$). The *chain* and the *fanout* nodes are assigned with descriptions of the *chain* and the *fanout links* in the results. The *semantic* nodes are used to facilitate *result/entity identification* in different versions of QURE-Pagelet. We now elaborate on these three types of nodes.

Fanout Node: The *fanout* node models a fanout of pages. The node should be assigned with the STALKER extraction rules for extracting the fanout links. This node should be nested at the *List* node, so that the rules for searching a fanout link are applied inside each element of this *List*. The *fanout* node does not appear in the output XML document. All the nodes that are nested at the *fanout* node appear nested at each element of the *List* which is the ancestor of the particular *fanout* node.

Chain Node: The *chain* node models a chain of pages. The node should be assigned with the STALKER extraction rules for extracting the chain links. The *chain* node should be nested at the *element* node so that the rules for searching a chain link are applied inside each next page with results. The *chain* node does not appear in the output XML document. All the nodes that are nested at the *chain* node appear nested at the element that is the ancestor of the particular *chain* node. There is also a parameter called *ChainType* that should be specified. This parameter can be assigned with only two possible values: "*RightChain*" or "*LeftChain*". We elaborate on this parameter below.

All the hidden web sites compose the chain links in their own way. The main type of chain link that is common for most of the sites is the link to the "next" page containing a set of results. For example, reconsider the `AutoTrader.com` query results. Figure 2 shows the first page with results. We can see that the "next" link is followed by the text "1" and the text is followed by 19 links to other pages containing results. The "next" in the second page (see Figure 7(a)) is followed by a link to the first page. This link is followed by the text "2", and the text is followed by 18 links to other pages containing results. And so on for all the pages in the *chain* except the last page (Figure 7(d)). Thus, the "next" is followed by different suffix in every page of the results except the last.

The STALKER extraction rules are rules for locating the beginning and the end, i.e., the prefix and suffix of the piece of data to be extracted. These rules may contain real tags and real text data or wildcards for them. In order to be extracted, a piece of information should be always surrounded by the same prefix and suffix. Some elements are surrounded by different prefixes and suffixes in different results, for these elements, several alternative prefixes and suffixes should be specified using sufficient number of learning examples. As we have noticed above, to extract a *chain* of results, we need to extract the "next" link from every page in the *chain*. Moreover, we have shown that for the "next" link it is common to be followed by (or follow) a block of links to every other page with results (see Figures 6(a) and (b)). Let us illustrate this issue with an example.

Example 3 Figure 7 shows the *chain* navigation links in several pages with results from `AutoTrader.com`. There are 20 pages totally in the *chain*. Figure 7(a) shows the first page with results. We can see that the "next" link is followed by the text "1" and the text is followed by 19 links to other pages with results. "Next" in the second page (see Figure 7(b)) is followed by one link (to the first page with results), that link is followed by the text "2", and the text is followed by 18 links to other pages with results and so on for all the pages in the *chain*. Thus, "next" is followed by a different suffix in every page of the results. According to STALKER technique, a user should provide 20 examples (one example per page from the chain) to learn extraction rules for the "next" link in the results from this site. ∎

As we can notice (see Figures 6), the "next" link along with the block of links to every other page with results are usually surrounded by the environment (decorative elements and links) that seems to be not changing through different pages with results. To decrease the number of learning examples for extracting the "next" link, we ask a user to specify learning examples for extracting the "next" link along with the block of links to every other page and also to specify whether the "next" link is followed by (see Figure 7(a)) or follows (see Figure 7(b)) the block of links to every other page. We call the first choice *LeftChain* and the second choice *RightChain*.

Semantic Node: Due to the dynamic nature of the hidden web, the underlying databases change at any time and in any way. Hence, the QURE-Pagelets returned by a specific query executed at different time points may change also. Often each query result in the QURE-Pagelet represents a distinct real world entity/object. For instance, each result in Figure 2 represents a car entity. Hence, it is necessary to be able to identify a particular entity in different versions of the query results to facilitate hidden web data integration, change detection, etc. For example, we may wish to determine whether the first car entity (`Ford F350 1 Ton Truck 4×4 Crew Cab`) in Figure 2 occurs in another version of the query results. The *semantic* node is used to capture necessary

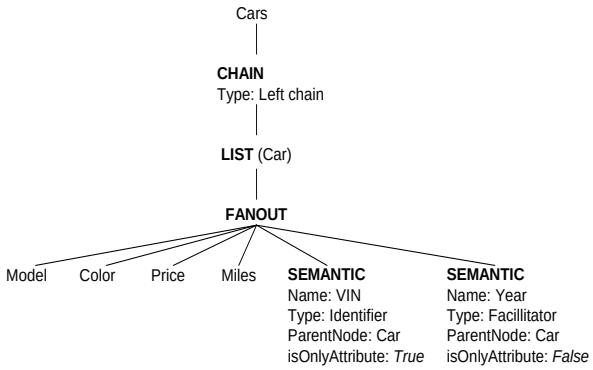


Fig. 8. $\mathcal{HW} - \mathcal{EC}$ description.

```
<Cars>
  <!-- Car 1.1 -->
  <Car Id="SAJDA42B52PA2375" F1="*">
    <Model>Jaguar XKR</Model>
    <Price>$63950</Price>
    <Miles>12150</Miles>
  </Car>
  <!-- Car 1.2 -->
  <Car Id="SAJDA42CO1NA17712" F1="2001">
    <Model>Jaguar XK8</Model>
    <Color>Platinum</Color>
    <Year>2001</Year>
    <Price>$49499</Price>
    <Miles>19586</Miles>
  </Car>
  <!-- Car 1.3 -->
  <Car F1="2001">
    <Model>Jaguar S-Type</Model>
    <Year>2001</Year>
    <Color>Bronze</Color>
    <Price>$40299</Price>
    <Miles>26567</Miles>
  </Car>
  <!-- Car 1.4 -->
  <Car F1="2000">
    <Model>Jaguar XKR</Model>
    <Year>2000</Year>
    <Color>Bronze</Color>
    <Price>$49595</Price>
    <Miles>30340</Miles>
  </Car>
</Cars>
```

(a) Version 1

```
<Cars>
  <!-- Car 2.1 -->
  <Car F1="2001">
    <Model>Jaguar XK8</Model>
    <Year>2001</Year>
    <Color>Bronze</Color>
    <Price>$50995</Price>
    <Miles>2650</Miles>
  </Car>
  <!-- Car 2.2 -->
  <Car Id="SAJDA42B52PA2375" F1="2002">
    <Model>Jaguar XKR</Model>
    <Year>2002</Year>
    <Color>Anthracite</Color>
    <Price>$63950</Price>
    <Miles>12150</Miles>
  </Car>
  <!-- Car 2.3 -->
  <Car Id="SAJDA42CO1NA17712" F1="2001">
    <Model>Jaguar XK8</Model>
    <Color>Platinum</Color>
    <Year>2001</Year>
    <Price>$49499</Price>
    <Miles>19586</Miles>
  </Car>
  <!-- Car 2.4 -->
  <Car F1="2000">
    <Model>Jaguar XKR</Model>
    <Year>2000</Year>
    <Color>Bronze</Color>
    <Price>$38995</Price>
    <Miles>1039</Miles>
  </Car>
</Cars>
```

(b) Version 2

Fig. 9. XML representation of query results.

information from the query results to facilitate such entity identification.

Semantic node is a leaf node in the $\mathcal{HW} - \mathcal{EC}$ tree. It is transformed to an XML attribute in the output XML file. It can be used to identify a result/entity in a QURE-Pagelet uniquely (*identifier*) or it may provide enough information to determine which results are related and has the potential to be identical (*facilitator*) in the old and new versions of the QURE-Pagelet. There are four parameters that needs to be specified for the semantic nodes in order to facilitate such entity identification and representation: *ParentNode*, *Type*, *Name*, and the *isOnlyAttribute*. The *ParentNode* parameter is the link to the particular node in the $\mathcal{HW} - \mathcal{EC}$ description that should be assigned with

this attribute in the output XML file. The *Type* parameter is for defining the type of an attribute (it can be of only two types: *identifier* or *facilitator*). We discuss these two types in the following subsections. The *Name* denotes the name of this node. The *isOnlyAttribute* contains a boolean value. If the *isOnlyAttribute* is set to "true" then it denotes that the piece of data extracted for this node should only appear in output XML as an attribute/value pair. Otherwise, it should appear both as an attribute and as an element. So if this information is needed as a part of an XML document then the *isOnlyAttribute* parameter is set to "false" so that the node appears as an element. Following is the example illustrating the $\mathcal{HW} - \mathcal{EC}$ formalism and mapping of a set of hidden web pages to XML.

Example 4 Consider the `AutoTrader.com`. Suppose a user wishes to search for *Jaguar* cars on 2nd and 5th July, 2003 respectively. The results are presented as a list of cars. Each result contains car details such as `Model`, `Year`, `Price`, `Color`, `Seller`, `Vehicle Identification Number (VIN)`, etc. Figure 8 depicts a *partial* $\mathcal{HW} - \mathcal{EC}$ tree for the QURE-pagelets and Figure 9 depicts the two versions of XML representation of the results according to the tree. For clarity and space, we only show a subset of the element set E in each result of the query. The root `Cars` node is established for uniting all the other nodes. The next node `Chain` models the set of pages connected with the "left" *chain* links. The `List(Car)` node is assigned with an iterative rule for extracting the elements of the `Car`. The `fanout` node denotes that each `Car` element contains a link to the page with extra data. The `fanout` is assigned with rules for extracting this link from the piece of HTML document that was extracted for each `Car` element in the previous step. The next level of the tree contains six elements. Four of them are `Element` nodes. The last two nodes are semantic nodes, each containing four parameters as discussed above. We can see in Figure 9 that the `VIN` is extracted only once for each `Car` as the value of attribute *Id* (*isOnlyAttribute* is set to `true`). It does not appear as a child element of a `Car` node. The rest of the elements are nested in the output XML the same way they are nested in the tree. ∎

3.3 Identifier

Some elements in a set of query results can serve as a unique identifier for the particular result, distinguishing them from other results. For example, the `Auction Id` uniquely characterizes every `Auction` information returned as the result of querying an on-line auction site. The `VIN` uniquely characterizes every `Car` in the query results from a car database. These elements are called *identifiers*. Table 1 contains examples of *Identifiers* in query results from some hidden web sites. An *identifier* may be either automatically generated by the hidden web database (like the `Auction Id`) or stored in the database along with the data (like the `VIN`). In this work we assume that the *identifier*,

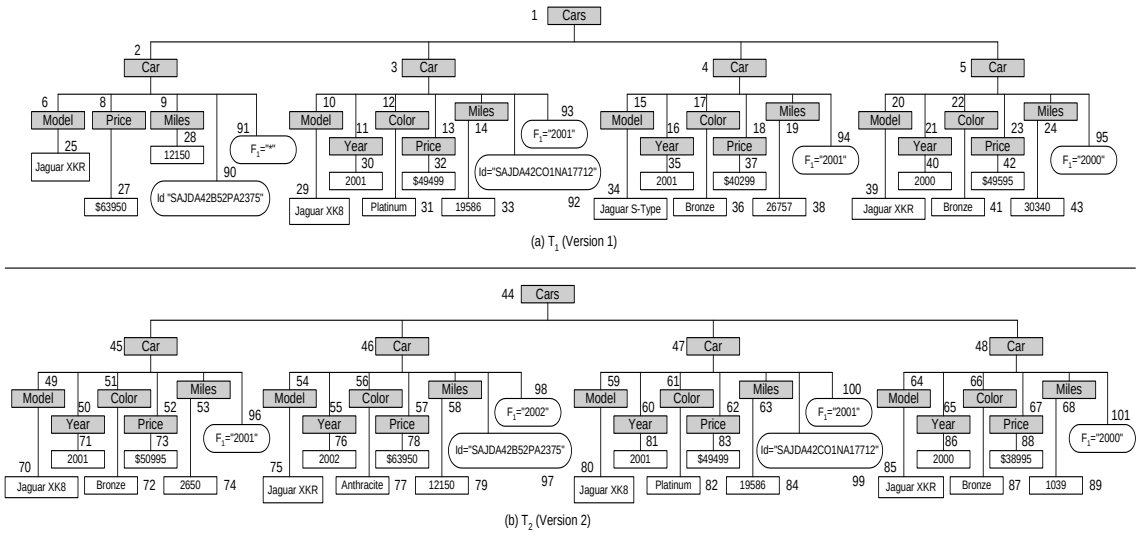


Fig. 10. Tree representation of two XML documents from Figures 9.

being assigned to a particular query result, does not change for this result through different versions of the query results. That is, an *identifier* behaves like an unique identifier or "key" for the result. However, it is possible for the *identifier* to be missing in a result. Also, if an *identifier* is specified (not specified) for a node in the initial version of the query results or when the node appeared for the first time in the results, then it will remain specified (not specified) throughout all versions, until the node is deleted. This reflects the case for most web sites we have studied. Note that we allow specifying only one *identifier* for each result. As each result is transformed into a subtree in the XML representation of the hidden web query results, we model the *identifier* of a particular node in the subtree as an XML attribute with name `Id` and the *identifier* information as value. We now illustrate with an example the usefulness of the *identifiers* in change detection. Note that details of the change detection problem is beyond the scope of this paper. The readers may refer to [13] for further details.

Example 5 Consider the QURE-pagelets of Autotrader (Figure 2). Given the query results and our understanding of its semantics, we may wish to state the following constraints:

- The `VIN` in the results uniquely identifies a particular car entity. Note that VIN may not be present for some results. For instance, the third car in the results depicted in Figure 9(a) has no `VIN` specified. However, if it exists then it will not be removed from the subsequent versions of query results involving the particular car.
- The `year` of manufacturing and the `model` of each car do not get modified in different versions. That is, a "mercedes" cannot be updated to "jaguar" or if the manufacturing year of a car is "2001" then it cannot be modified

Site	Query	Element	Identifier	Presence
www.autotrader.com (Buying cars, etc.)	Search for cars	Car (siblings: Model, Color, Year, etc.)	VIN (Vehicle Identification Number)	Optional
www.autobytel.com (Buying cars, etc.)	Search for cars	Car (siblings: Model, Color, Year, etc.)	VIN (Vehicle Identification Number)	Optional
www.travelocity.com (Booking hotels, flights, cars, etc.)	Search for flights	Flight (siblings: Departure, Arrival, Airline, etc.)	Flight Number	Compulsory
www.ebookers.com (Booking hotels, flights, cars, etc.)	Search for flights	Flight (siblings: Departure, Arrival, Airline, etc.)	Flight Number	Compulsory
libweb.ntu.edu.sg (NTU library)	Search for titles	Title (siblings: Status, Authors, Location)	Call Number	Compulsory
www.pubmed.org (Medical publications)	Search for publications	Publication (siblings: Title, Authors, Date, etc.)	PMID (PubMed ID)	Compulsory
www.auctions.yahoo.com (On-line auctions)	Search for auctions	Auction (siblings: Title, Seller, Price, etc.)	Yahoo! Auction ID	Compulsory
www.ebay.com (On-line auctions)	Search for auctions	Auction (siblings: Title, Seller, Price, etc.)	Ebay Auction ID	Compulsory
www.nationjob.com (Job database)	Search for jobs	Job (siblings: Description, Requirements, Contact, etc.)	NationJob Job ID	Compulsory
ads.harvard.edu (Database of astronomy abstracts)	Search for abstracts	Abstract (siblings: Authors, Journal, Date, etc.)	NASA Bibliographic Code	Compulsory
www.uspto.gov (US Patent Full Text Database)	Search for patents	Patent (siblings: Patent Number, Title, etc.)	Patent Number	Compulsory
www.architectureweek.com (Architecture magazine)	Search for articles	Article (siblings: Authors, Issue, Pages, etc.)	Article ID	Compulsory

Table 1
Examples of *Identifiers* in query results from different hidden web sites.

to "2002" or any other year in the subsequent versions.
- Similarly, the `seller` attribute of a car does not get modified in different versions for the same car entity.

In particular, the `VIN` is the unique identifier for a specific car. If we use the VINs as *identifiers* for the cars in the list, then we can distinguish cars with VINs effectively. A sample of tree representations of the documents in Figure 9 are shown in Figure 10. The `Car` nodes in T_1 and T_2 have child attributes with name `Id` and value equal to the `VIN`. Intuitively, if we wish to detect the changes between the two versions of the query results, then we can match the `Car` nodes between two subtrees by comparing the `Id` values. For instance, the node 2 in T_1 matches the node 46 in T_2 and the node 3 in T_1 matches the node 47 in T_2 (same `VIN` values). However, the nodes 2 and 3 do not match the node 45 as it does not have any *identifier* attribute. ∎

Site	Query	Result Item	Facilitator	Presence
www.autotrader.com (Buying cars, etc.)	Search for cars	Car (siblings: Model, Color, Year, etc.)	Seller	Compulsory
	Search for cars	Car (siblings: Model, Color, Year, etc.)	Year	Optional
www.cbooks.com (buying computer books, etc.)	Search for books	Book (siblings: Author, Title, Price, etc.)	Title	Compulsory
	Search for books	Book (siblings: Author, Title, Price, etc.)	Publisher	Optional
www.imdb.com (information on movies, actors, etc.)	Search for movies	Movie (siblings: Title, Status, Year, Crew, etc.)	Title	Compulsory
	Search for movies	Movie (siblings: Title, Status, Year, Crew, etc.)	Status	Optional
www.travelocity.com (booking hotels, flights, cars, etc.)	Search for hotels	Hotel (siblings: Title, Address, Rooms, Rates, etc.)	Title	Compulsory
	Search for cars to rent	Car (siblings: Model, Rental price, Agency, etc.)	Agency	Compulsory
www.stanford.edu	Search for people in Stanford	Person (siblings: Name, Status, Department, etc.)	Designation	Compulsory
	Search for people in Stanford	Person (siblings: Name, Status, Department, etc.)	Name	Compulsory
star-www.rl.ac.uk ("Persons in Astronomy" database)	Search for astronomy related persons	Person (siblings: Name, Institution, E-mail, etc.)	Name	Compulsory
www.hoovers.com (Hoover's – company intelligence)	Search for companies	Company (siblings: Title, Address, Phone, Company News, etc.)	Title	Compulsory
	Search for companies	Company (siblings: Title, Address, Phone, Company News, etc.)	Phone	Optional
www.ipl.org (IPL Association Finder)	Search for associations	Association (siblings: Title, Description, URL, etc.)	Title	Compulsory
www.kiplinger.com (Kiplinger financial publications)	Search for publications	Publication (siblings: Title, Date, Abstract, Number of words, etc.)	Date	Optional

Table 2
Examples of *Facilitators* in query results from different hidden web sites.

3.4 Facilitator

One or more elements in the result of the hidden web query result set can serve as non-unique characteristics for distinguishing the results from one another. This is particularly important when the results do not have any *identifier* attribute. Two results that have the same characteristics (same attribute/value pair) can be matched with each other. While results that have different characteristics can not be matched with each other. Examples of types of such characteristics are: the `Year` or `Model` of a `Car` node in the query results from

car trading site, the `Title` or `Year` for a `Movie` node in the query results from movie database, the `Title` or `Publisher` for a `Book` node in the query results from an online book catalog. These non-unique elements are called *facilitators*. Note that these elements may not identify a result (entity) uniquely. But they may provide enough information to identify results that do not refer to the same entities. For example , if we wish to match the last result (node 5) in Figure 10(a) with the first result (node 45) in Figure 10(b) then these two cars are not same entities in the two versions as they have different manufacturing `year`.

We allow specifying any number of *facilitators* on a node. The *facilitators* are denoted by node attributes with names F_1, F_2, ... , F_n for all n *facilitator* attributes specified for a particular node. If a node does not have a *facilitator* attribute (the subelement may be missing) then the *facilitator* value is set to "`*`". Note that the *facilitator* attribute for a node can appear in any version of the query results, but once it appears we assume that it will not disappear in the future versions. As we never know which *facilitator* may appear for a node in the future, a node with missing *facilitator* attribute should be matched with nodes having *facilitators*. This statement reflects the case for most hidden web sites we have studied. Table 2 depicts some examples of *facilitators* in query results from different hidden web sites. In particular, this table shows us that the presence of *facilitators* is not always compulsory.

The choice of *facilitators* made by the user is based on the following two guidelines.

(1) Let $R_i(A_k)$ and $R_i(V_k)$ be the attribute set and its corresponding value set in a hidden web query result R_i. Let R_1 and R_2 be two query results for a given query. If $R_1(V_k) \neq R_2(V_k)$ indicates that R_1 and R_2 represents two distinct objects then A_k can be considered as a possible *facilitator*. Note that, unlike *identifiers*, if $R_1(V_k) = R_2(V_k)$ then it *does not necessarily* indicate that R_1 and R_2 represents two identical objects. For example, consider the `year` of manufacturing of a car. If two query results have different manufacturing year then definitely the two cars cannot be identical. On the other hand, if the manufacturing years are identical then it does not necessarily indicate that the cars are identical. However, consider the `price` attribute of a car. If the price values of two results are distinct then it does not necessarily indicate that they represent two distinct car objects as the price of the car can be updated. Hence, choosing `year` of manufacturing over `price` as *facilitator* is a better choice as it can potentially reduce number of comparisons required to identify identical car entities/objects.

(2) Let A_{k1}, A_{k2}, ..., A_{kn} be the set of possible *facilitators* identified from the above step and C_1, C_2, ..., C_n be the number of distinct possible values of the *facilitators* in the query results for $n > 1$. Then, A_{ki} is chosen as

the *facilitator* if $C_i \geq C_j$ for $j \neq i$ and $\forall\ 0 < j \leq n$. Intuitively, we choose the *facilitator* that has largest number of distinct values. This is because the larger the number of distinct values the potentially lesser number to results that needs to be compared to identify identical/related objects. For example, consider the attributes `year` of manufacturing and `number of doors` of a car object. Note that if the values of these attributes are not identical in a pair of query results then definitely they represent distinct car objects. Generally, the potential number of distinct values for `year` is much larger than that of `number of doors`. Hence, if we use `year` rather than `number of doors` as *facilitator*, then matching by *facilitators* becomes more effective as the number of object comparison will be much lesser.

We now illustrate with an example how the *facilitators* can be useful for the change detection problem.

Example 6 Reconsider the Figure 9. We can find several candidates for *facilitators*, i.e., `Color`, `Year`, or `Model`. However, based on the above guidelines, it is reasonable to use the `Year` or `Model` as the *facilitator*. Figure 10 shows the *facilitators* for various nodes. There is one *facilitator* specified: the `Year` as an attribute with name F_1 for every `Car` node. Note that if a `Car` node does not have an subelement `Year` then F_1 is set to "`*`". Now let us match the node 45 in T_2 with all the nodes in T_1. Observe that node 45 does not have a `VIN`. Therefore, it cannot match with nodes 2 and 3. Hence we do not need to compare node 45 with these nodes. Using F_1 we also observe that the node 45 cannot match with node 5 as the facilitators do not match. We can see that the node 45 only matches node 4 in T_1 as it does no have any `VIN` and its F_1="2001". However, this is not sufficient information to confirm whether these two nodes represent the same car entity. But if we use both the `Year` and `Model` as facilitators then we can answer this question by comparing the `Model` of node 45 with that of node 4. As the `Model` of nodes 4 and 45 are not identical, we can say that these nodes do not represent the same car entity. Thus, we can state that the node 45 is inserted in T_2 as none of the car entities in T_1 matches the car entity described by node 45. ∎

4 Algorithm HW-Transform

Figure 11 depicts the *HW-Transform* algorithm for transforming QURE-pagelets to XML format. This algorithm takes as input the first page of the results and the $\mathcal{HW} - \mathcal{EC}$ description of these results, and returns the XML representation of the results. There are three main steps in the algorithm. The first step is to **extract information** from query results and generate tree T storing extracted information in hierarchical order according to the $\mathcal{HW} - \mathcal{EC}$ description of these results. Lines 8-11 in Figure 11 describe this step. In the tree T, all the attributes are presented as leaf nodes, according to the $\mathcal{HW} - \mathcal{EC}$

```
Input: Index, /* index page of query results */
       HW-EC  /* HW-EC description of these results */
Output: Doc   /* XML representation of query results */


 1  T: tree    /* for storing tree representation of
                  query results */
 2  Ta: tree  /* which is enabled to store attributes
                 for every node */
 3  [Te]: set of trees
 4  Doc: XML document
 5  set T, Ta, [Te], and Doc empty
 6  let Root be the root of HW-EC
 7  add Root as a root to T
    /* extract information from query results */
 8  for all Ch that is a child of Root in HW-EC do
 9    add ExtractNode(Index, Ch) to [Te]
10  end for
11  add every Te from [Te] as a child of root node to T
    /* assign attributes */
12  Ta = AssignAttributes(T, HW-EC)
    /* generate XML */
13  Doc = GenerateXML(Ta)
14  Return Doc
```

Fig. 11. The algorithm **HW-Transform**.

```
Input: Doc,   /* a piece of source code */
       HW-EC, /* HW-EC description of query results */
       N  /* node in HW-EC to be extracted from Doc */
Outoput: [Te] /* a set of trees */

 1  [Te]: set of trees
 2  set [Te] empty
    /* extract Element or Attribute */
 3  if (N.Type=="Element" or N.Type=="Attribute")
 4    add ExtractElement(Doc, HW-EC, N) to [Te]
 5    Return [Te]
 6  end if
    /* extract List */
 7  if (N.Type=="List")
 8    add ExtractList(Doc, HW-EC, N) to [Te]
 9    Return [Te]
10  end if
    /* extract Chain */
11  if (N.Type=="Chain")
12    repeat
13      for all Ch that is a child of N in HW-EC do
14        add ExtractList(Doc, HW-EC, Ch) to [Te]
15      end for
        /* load next page in the chain using rule
           assigned to N to extract its URL */
16      Doc = Doc.NextPage
17    until (Doc.NextPage==∅)
18    Return [Te]
19  end if
    /* extract Fanout */
20  if (N.Type=="Fanout")
      /* load fanout page in the chain using rule
         assigned to N to extract its URL */
21    Doc = Doc.GetFanoutPage
22    for all Ch that is a child of N in HW-EC do
23      add ExtractNode(Doc, HW-EC, Ch) to [Te]
24    end for
25    Return [Te]
26  end if
```

Fig. 12. The algorithm **ExtractNode**.

description. The next step is to traverse tree T in order to **assign attributes** to the nodes. Line 12 in Figure 11 describe this step. The final step is to **generate XML representation of the results**. Line 13 in Figure 11 describe this step. We elaborate on these steps now.

4.1 Extracting Information

ExtractNode is a recursive algorithm for extracting pieces of data from query results corresponding node N in the $\mathcal{HW} - \mathcal{EC}$ description of the results. As output, **ExtractNode** returns a set (one or more) of trees representing com-

```
Input: T,    /* tree representation of query results */
       HW-EC /* HW-EC description of results */
Output: Ta  /* tree with attributes assigned to nodes */

 1  Ta: tree /* which is enabled to store attributes for every node */
 2  M, N, A1, A2 ... An: nodes in HW-EC
 3  p: integer
 4  Ta = T
 5  for all N in HW-EC which has at least one attribute defined do
 6    let A1, A2 ... An denote all the nodes that are attributes for N
 7    if (N is the ancestor for all its attributes in HW-EC)
 8      M = N
 9    else
10      let M be the least common ancestor of N, A1, A2 ... An
11    end if
12    for all Node of type M in Ta do
13      p=1
        /* extract Attribute */
14      for all Ak, 1 <= k <= n do
15        if (there is node Attr of type Ak
                 in subtree of Ta rooted at Node)
            /* assign Identifier attribute */
16          if (Attr.AttributeType=="Identifier")
17            add attribute with name Id and
                 value Hash(Attr.Value) to Node
18          if (Ak.isOnlyAttribute) then delete Attr from Ta endif
          /* assign Facilitator attribute */
19        else if(Attr.AttributeType=="Facilitator")
20          add attribute with name "F_"+p and
                 value Hash(Attr.Value) to Node
21          if (Ak.isOnlyAttribute) then delete Attr from Ta endif
22          p=p+1
          /* assign common attribute */
23        else
24          add attribute with name Attr.Name and
                 value Attr.Value to Node
25          if (Ak.isOnlyAttribute) then delete Attr from Ta endif
26        end if
        /* substitute Facilitator attribute that is not found */
27      else if(Attr.AttributeType=="Facilitator")
28        add attribute with name "F_"+p and value "*" to Node
29        p=p+1
30      end if
31    end for
32  end for
33 end for
34 Return Ta
```

Fig. 13. The algorithm **AssignAttributes**.

plete set of data corresponding to node N extracted from the query results. In this data set Attribute nodes are extracted as Element nodes.

The **ExtractNode** algorithm is based on functions **ExtractList** and **ExtractElement** (see Section 2) designed for iterating Lists and extracting Elements according $\mathcal{EC}$ description of the page. The **ExtractNode** algorithm is designed to provide the superstructure that enables us to work with a set of pages. So all the procedures used in this algorithm are based on **ExtractList** and **ExtractElement**.

Let us now go to Figure 12 showing the **ExtractNode** algorithm. Lines 1,2 contain initialization of the set trees T_e that is to be grown by further steps and finally returned as output. Lines 3-6 contain procedures that will be executed if input node is *Element* or *Attribute*. Lines 7-10 contain procedures that should be executed if input node is *List*. Lines 11-19 contain procedures that will be executed if input node is *Chain*. These procedures include downloading all the pages from a *Chain*. Lines 20-26 contain procedures that should be executed if input node is *Fanout*. These procedures include downloading a *Fanout* page. If a particular *List* or *Element* extracted by **ExtractList** or **ExtractElement** is still the ancestor of any *Chains* or *Fanouts*, **ExtractNode** is called recursively.

4.2 Assigning Attributes

In the tree T, all the attributes are presented as leaf nodes, according to the $\mathcal{HW} - \mathcal{EC}$ description. The next step is traverse tree T in order to assign attributes to the nodes. Line 12 in Figure 11 describes this step. Note that

we do not assign attributes to particular nodes while first parsing the tree in the previous step as attributes can be located in different parts of the tree. It is faster to assign all attributes doing one traversal in a separate step than doing a lot of additional traversals in the previous step. Figure 13 shows the algorithm for assigning attributes.

The algorithm traverses tree T generated by the previous step and assigns attributes in tree T according to the $\mathcal{HW} - \mathcal{EC}$ description of the query results. First, the algorithm localizes the least common ancestor of the node and its attribute, and then localizes the attribute by the relative path from this ancestor. This procedure runs until all the nodes that are assigned with any attributes in the $\mathcal{HW} - \mathcal{EC}$ description are traversed in this manner.

Let us now go through the algorithm (Figure 13) step by step. Lines 1-4 contain initialization and description of the variables used in the algorithm. Lines 5-33 contain the main cycle going through all the nodes that have any attributes assigned in the $HW - EC$. The embedded cycle in lines 12-32 goes through last common ancestors of such nodes and their attributes. Inside this cycle there are three conditions: for assigning *Identifier* attributes in lines 16-18, for assigning *Facilitator* attributes in lines 19-22, and for assigning common attributes in lines 23-26. The last action inside this cycle is filling values of *facilitator* attributes that are not found with "$*$".

The final step is to **generate XML representation of the results**. To generate XML document from the tree, we use simple depth/breadth-first traversal of tree starting from the root.

5 Implementation

We used Microsoft Windows 2000 Professional as operating system. We have implemented HW-STALKER using Java. We first briefly describe our system architecture and then present some performance results.

5.1 Architecture

As shown in Figure 14, our prototype system consists of two main modules namely *Extraction Module* and *Change Detection Module*, and *Repository* for storing intermediate data and system output.

- **Extraction Module**. This module is designed for extracting information from original HTML-formatted hidden web query results and for transforming it into XML format. XML versions of query results are to be stored in *Repository*. There are three inner modules in *Extraction Module*. *User In-*

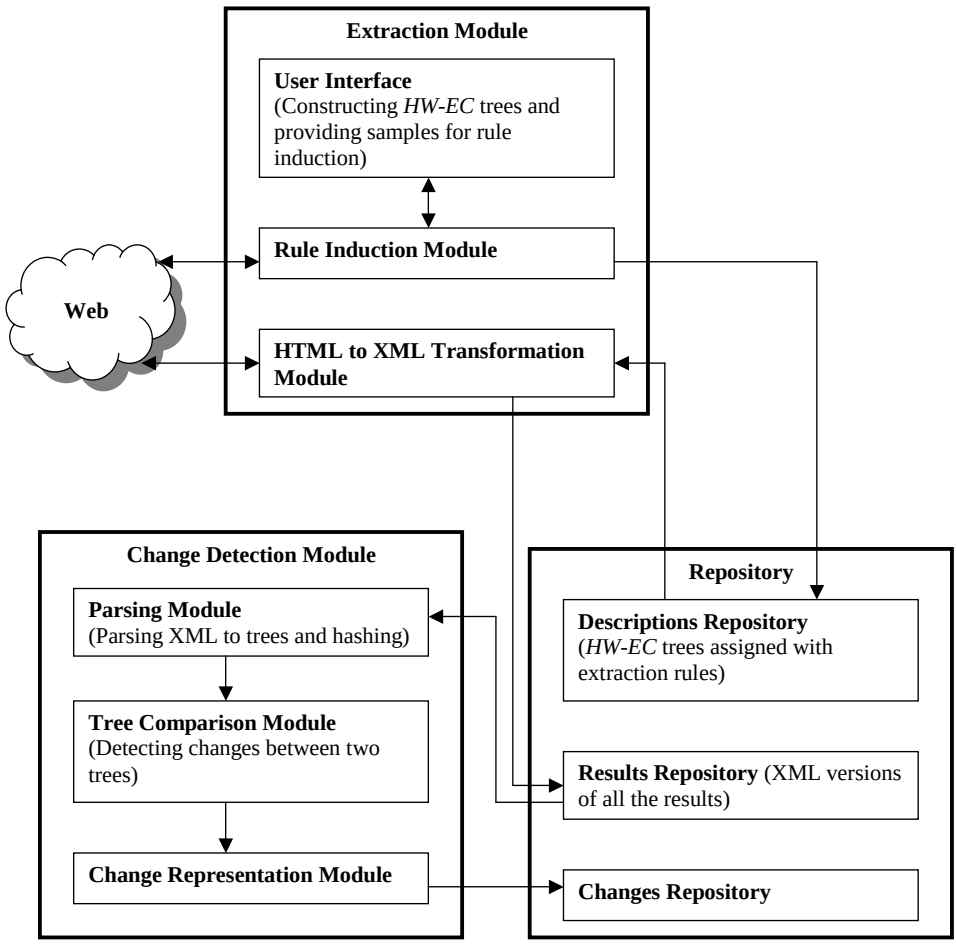


Fig. 14. The system architecture.

terface is designed to assist a user in constructing $\mathcal{HW}-\mathcal{EC}$ tree for a set of pages with hidden web query results. *User Interface* is also designed for marking examples that are further used for rule induction. The resulting tree is to be stored in *Repository*. *Rule Induction Module* is designed to induce extraction rules being provided with the examples mapped by a user and $\mathcal{HW}-\mathcal{EC}$ tree constructed by a user that are both stored in the *Repository*. Rule induction is based on the STALKER algorithm as discussed earlier. *HTML to XML Transformation Module* is designed to transform QURE-pagelets into XML format being provided with extraction rules $\mathcal{HW}-\mathcal{EC}$ tree that are taken from the *Repository*. This module is based on the *HW-Transform* algorithm (Section 4).

- **Change Detection Module**. This module is designed to detect changes between different versions of query results. It is based on the *HW-Diff* algorithm [13]. Note that this module is beyond the scope of this paper. The XML versions of these changes are to be stored in *Repository*. There are three inner modules in *Change Detection Module*. *Parsing Module* is designed for converting XML to trees and hashing tree nodes. *Tree Com-*

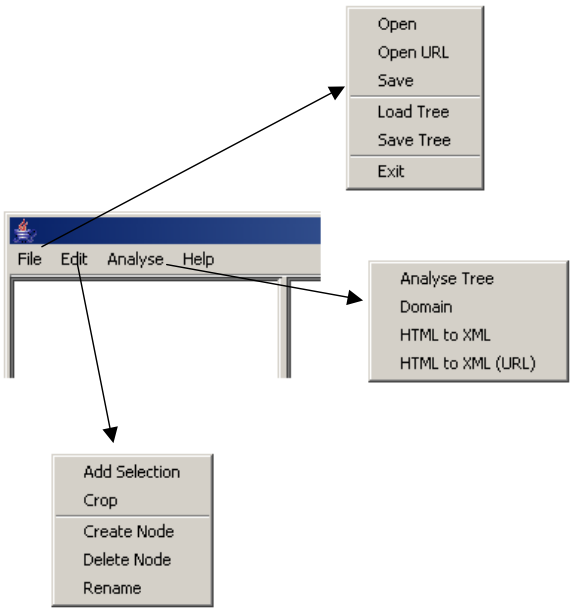
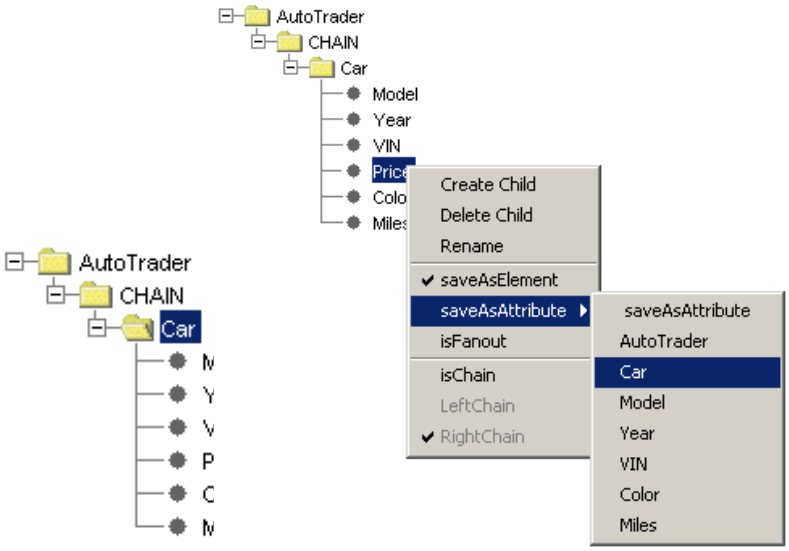


Fig. 15. Main menus of User Interface from Extraction Module.

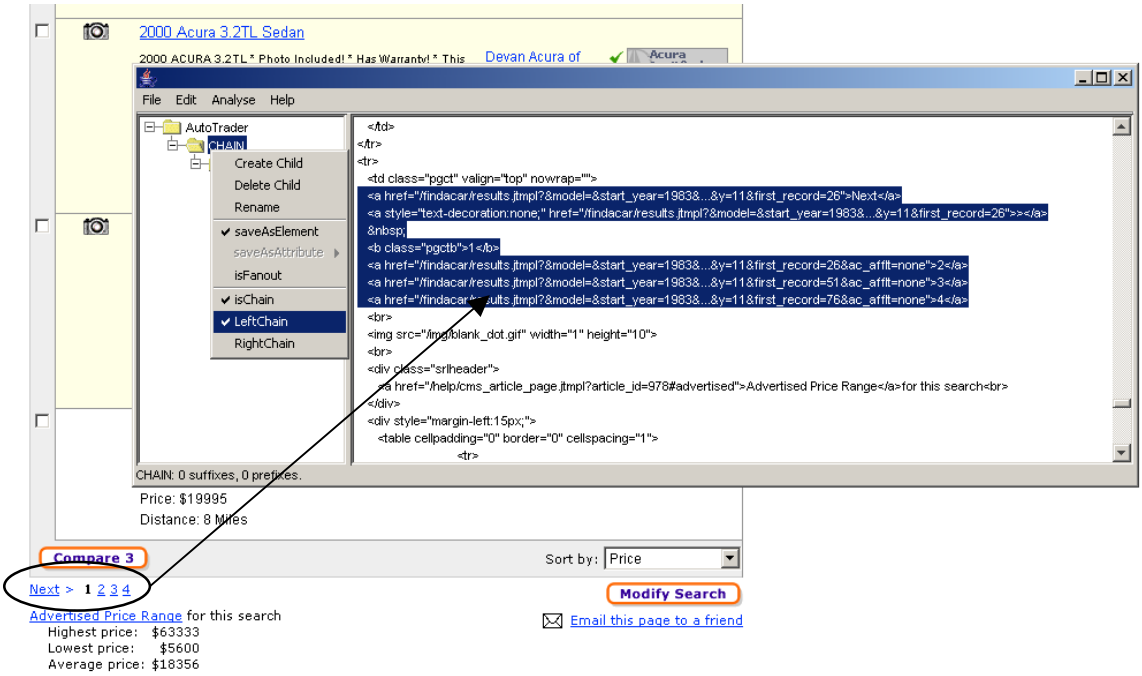


(a) Results general structure. (b) Assigning attributes.

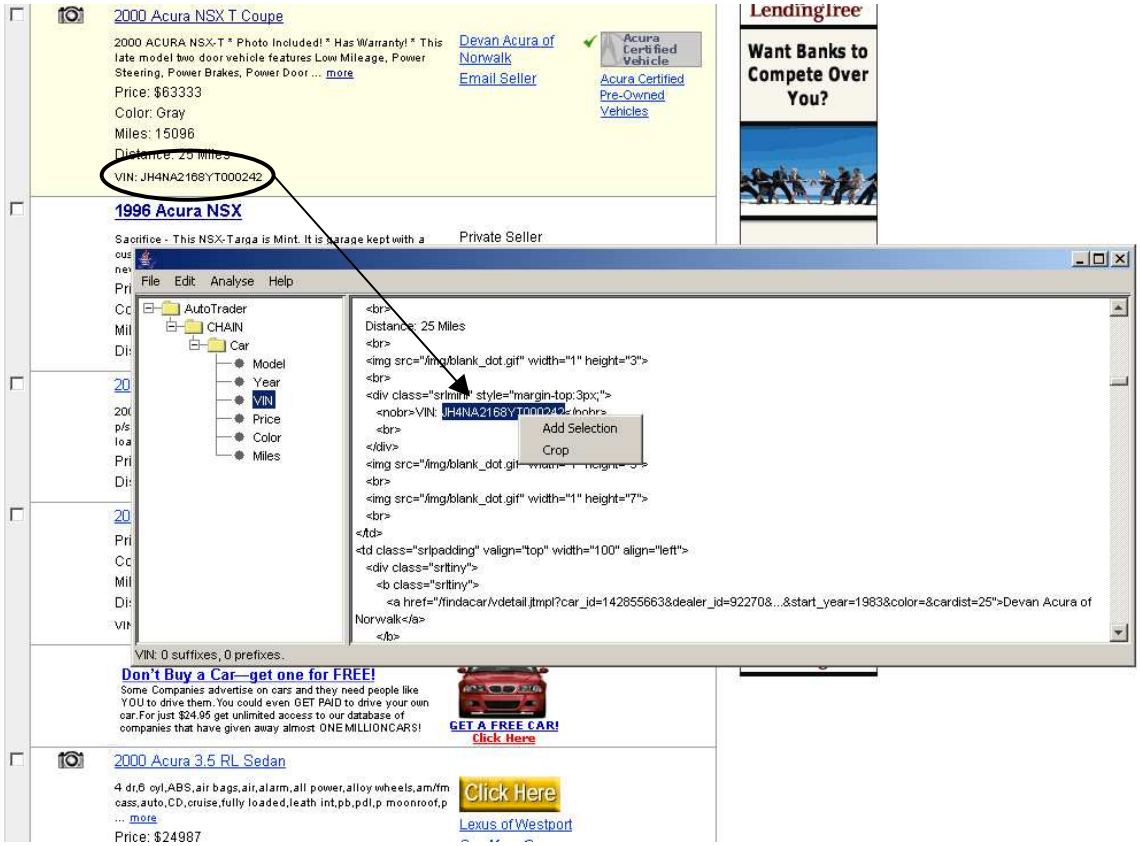
Fig. 16. Constructing $\mathcal{HW} - \mathcal{EC}$ description of results from `AutoTrader.com`.

parison Module is designed for detecting changes between two trees representing query results. *Change Representation Module* is designed to convert the changes detected between two trees to XML format.

- **Repository**. Repository is aimed to store $\mathcal{HW} - \mathcal{EC}$ trees, XML versions of query results, and the changes that are detected to hidden web data. In the current implementation *Repository* is simply based on file system.



(a) Chain selection.



(b) Element selection.

Fig. 17. Sample selections in results from `AutoTrader.com`.

Let us now illustrate the transformation of QURE-pagelets to XML step by step.

(1) The first step is to specify $\mathcal{HW} - \mathcal{EC}$ tree describing query results from this site. The User Interface in our system has two main areas. One area is for composing $\mathcal{HW} - \mathcal{EC}$ trees and other area is for marking training examples. Main menu of User Interface is depicted in Figure 15. Figure 16 shows a tree composed to describe query results from `AutoTrader.com`. This figure also shows how a user can assign some node in $\mathcal{HW} - \mathcal{EC}$ tree as an attribute of another node.
(2) The next step is to induce extraction rules by studying examples of queries. Figures 17 shows examples of assigning training examples to the nodes of the $\mathcal{HW} - \mathcal{EC}$ tree composed at the previous step. After all the examples are assigned, a user starts rule induction process by going to $Analyse \rightarrow AnalyseTree$ in main menu (see Figure 15). After all the rules are induced, a user can save $\mathcal{HW} - \mathcal{EC}$ tree with assigned rules and use this tree later to extract query results.
(3) Finally, a user uses a $\mathcal{HW} - \mathcal{EC}$ created at the previous step to transform the QURE-pagelets to XML and stores them in the Repository. At this step a user uses the same User Interface (See the Main menu from Figure 15: $File \rightarrow LoadTree$ and $Analyse \rightarrow HTMLtoXML$).

5.2 Performance Study

In this section, we present performance analysis of our prototype system. All the experiments have been performed on a Pentium 4 CPU 2.4 GHz with 512 MB of RAM. We use the data from the following six hidden web sites for our experiments: `AutoTrader.com`, `Amazon.com`, `Architecture.com`, `NationJob.com`, `IMDb.com`, and `CiteSeer.org`.

5.2.1 Extraction Time

To evaluate the performance of the extraction of relevant data from hidden web pages to XML format, we have to evaluate the performance of rule induction mechanism and the performance of HTML to XML transformation mechanism. Performance of the rule induction system is determined by the time a user spends on creating $\mathcal{HW} - \mathcal{EC}$ tree. It is also determined by the time a user needs to provide examples of data for each element of $\mathcal{HW} - \mathcal{EC}$ tree. This time dramatically depends on the number of examples a user should provide for each element of results to learn correct extraction rules. The $\mathcal{HW} - \mathcal{EC}$ trees that were composed to describe results of sample sites are presented in Figure 18. The variable Pec_i denotes the facilitator in this figure. The number of training examples for some of the elements of the hidden web data that was used in our experiments are shown in Table 3. Observe that we only need one example to learn extraction rules for one element for 40% of elements in the

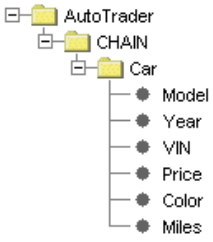
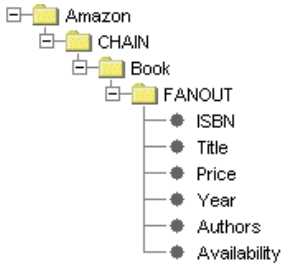
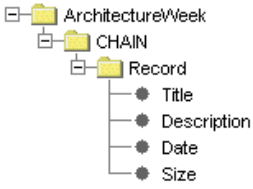
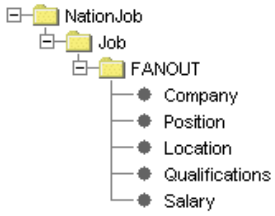
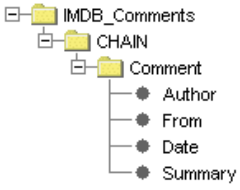
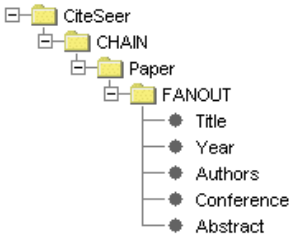
AutoTrader.com Searching for cars with wide range of car parameters http://www.autotrader.com/findacar/index.jtmpl?ac_afflt=none		<!ELEMENT AutoTrader (Car)*> <!ELEMENT Car (Model, Year, VIN, Price, Color, Miles)> <!-- VIN --> <!ATTLIST Car Id CDATA #IMPLIED> <!-- Model --> <!ATTLIST Car Pec_1 CDATA #IMPLIED> <!-- Year --> <!ATTLIST Car Pec_2 CDATA #IMPLIED> <!-- Color --> <!ATTLIST Car Pec_3 CDATA #IMPLIED> <!ELEMENT Model (#PCDATA)> <!ELEMENT Year (#PCDATA)> <!ELEMENT VIN (#PCDATA) > <!ELEMENT Price (#PCDATA)> <!ELEMENT Color (#PCDATA)> <!ELEMENT Miles (#PCDATA)>
Amazon.com Searching for books with keyword http://www.amazon.com		<!ELEMENT Amazon (Book)*> <!ELEMENT Book (ISBN, Title, Price, Year, Authors, Availability)> <!-- ISBN --> <!ATTLIST Book Id CDATA #IMPLIED> <!-- Year --> <!ATTLIST Book Pec_1 CDATA #IMPLIED> <!ELEMENT ISBN (#PCDATA)> <!ELEMENT Title (#PCDATA)> <!ELEMENT Price (#PCDATA)> <!ELEMENT Year (#PCDATA)> <!ELEMENT Authors (#PCDATA)> <!ELEMENT Availability (#PCDATA)>
ArchitectureWeek.com Searching for exhibitions, references, articles, etc. with keyword http://www.architectureweek.com/search.html		<!ELEMENT ArchitectureWeek (Record)*> <!ELEMENT Record (Title, Description, Date, Size)> <!-- Description --> <!ATTLIST Record Id CDATA #IMPLIED> <!-- Date --> <!ATTLIST Record Pec_1 CDATA #IMPLIED> <!-- Size --> <!ATTLIST Record Pec_2 CDATA #IMPLIED> <!ELEMENT Title (#PCDATA)> <!ELEMENT Description (#PCDATA)> <!ELEMENT Date (#PCDATA)> <!ELEMENT Size (#PCDATA)>
NationJob.com Searching for jobs with wide range of parameters http://www.nationjob.com		<!ELEMENT NationJob (Job)*> <!ELEMENT Job (Company, Position, Location, Qualifications, Salary)> <!-- Company --> <!ATTLIST Job Pec_1 CDATA #IMPLIED> <!-- Position --> <!ATTLIST Job Pec_2 CDATA #IMPLIED> <!ELEMENT Company (#PCDATA)> <!ELEMENT Position (#PCDATA)> <!ELEMENT Location (#PCDATA)> <!ELEMENT Qualifications (#PCDATA)> <!ELEMENT Salary (#PCDATA)>
IMDb.com Searching for user comments with movie title http://www.imdb.com		<!ELEMENT IMDB_Comments (Comment)*> <!ELEMENT Comment (Author, From, Date, Summary)> <!-- Author --> <!ATTLIST Comment Pec_1 CDATA #IMPLIED> <!-- Date --> <!ATTLIST Comment Pec_2 CDATA #IMPLIED> <!ELEMENT Author (#PCDATA)> <!ELEMENT From (#PCDATA)> <!ELEMENT Date (#PCDATA)> <!ELEMENT Summary (#PCDATA)>
CiteSeer.org Searching for on-line scientific papers with keyword http://citeseer.nj.nec.com/cs		<!ELEMENT CiteSeer (Paper)*> <!ELEMENT Parper (Title, Year, Authors, Conference, Abstract)> <!-- Abstract --> <!ATTLIST Paper Id CDATA #IMPLIED> <!-- Title --> <!ATTLIST Comment Pec_1 CDATA #IMPLIED> <!-- Year --> <!ATTLIST Comment Pec_2 CDATA #IMPLIED> <!-- Authors --> <!ATTLIST Comment Pec_3 CDATA #IMPLIED> <!ELEMENT Title (#PCDATA)> <!ELEMENT Year (#PCDATA)> <!ELEMENT Authors (#PCDATA)> <!ELEMENT Conference (#PCDATA)> <!ELEMENT Abstract (#PCDATA)>

Fig. 18. $\mathcal{HW} - \mathcal{EC}$ descriptions and DTDs for modelling query results from different sites.

Site	Element	Number of samples	Site	Element	Number of samples
AutoTrader.com	AutoTrader	1	Amazon.com	Amazon	1
	Car	8		Book	6
	Model	5		ISBN	2
	Year	5		Title	1
	VIN	1		Price	1
	Price	1		Year	1
	Color	1		Authors	3
	Miles	2		Availability	1
	CHAIN	5		CHAIN	6
				FANOUT	3
ArchitectureWeek.com	ArchitectureWeek	2	NationJob.com	NationJob	1
	Record	4		Job	2
	Title	2		Company	1
	Description	2		Position	1
	Date	3		Location	2
	Size	2		Qualifications	2
	CHAIN	3		Salary	3
				FANOUT	1
IMDb.com	IMDB_Comments	2	CiteSeer.org	CiteSeer	2
	Comment	3		Parper	3
	Author	2		Title	1
	From	2		Year	1
	Date	1		Authors	1
	Summary	1		Conference	1
	CHAIN	3		Abstract	1
				CHAIN	4
				FANOUT	3

Table 3
Number of samples needed to learn extraction rules.

query results. We need more than five examples for one element for only 5% of the elements. The number of results needed to learn extraction rules for particular element is determined by the number of different HTML-environments which can be found for this element in different results [11].

To evaluate the performance of the Transformation Module that is based on the *HW-Transform* algorithm, we have extracted the results of different queries from the six hidden web sites. The list of queries that we have used in this experiment is shown in Table 4. The complete results of this experiment is shown in Table 5. The summary of this experiment is shown in Figure 19. This figure demonstrates us that the dependence between extraction time and number of extracted results can be approximated as linear function.

Site	Query	Number of results	Number of CHAIN pages	Number of FANOUT pages
AutoTrader.com	2000-2004 Acura of any model within 25 miles from ZIP 00501	48	2	-
	1983-2004 Ford Escort within 25 Miles from ZIP 10001	120	5	-
	1983-2004 Jaguar of any model within 50 Miles from ZIP 00501	202	9	-
	1983-2004 Land Rover Range Rover within 200 Miles from ZIP 00501	310	13	-
	1990-2004 Cadillac with mileage under 75,000 within 50 Miles from ZIP10001	430	18	-
	1991-2004 Toyota with price range from 10,000 to 15,000 within 50 Miles from ZIP 10001	472	19	-
	1983-2004 Ford of any model within 25 Miles from ZIP 10001	500	20	-
Amazon.com	Search for "gardenia"	21	2	21
	Search for "snooker"	190	19	190
	Search for "intranet"	431	44	431
	Search for "dock"	599	60	599
	Search for "dot"	1001	101	1001
ArchitectureWeek.com	Search for "chalet"	35	4	-
	Search for "tall building"	311	32	-
	Search for "column"	619	62	-
	Search for "street"	780	78	-
	Search for "environment"	1053	106	-
NationJob.com	Computers/I.T./Telecommunications: Computer Operator	10	-	10
	Computers/I.T./Telecommunications: Software Design/Project Management	121	-	121
	Accounting/Finance/Insurance: Banking	693	-	693
	Education/Teaching/Child Care	1478	-	1478
IMDb.com	User comments to "Arrival"	58	3	-
	User comments to "Once Upon a Time in Mexico"	201	11	-
	User comments to "Star Wars V"	746	38	-
	User comments to "Harry Potter and the Sorcerer's Stone"	1212	61	-
CiteSeer.org	Search for "hidden web"	38	2	38
	Search for "google"	523	27	523
	Search for "p2p"	754	38	754
	Search for "web"	1000	50	1000

Table 4
Different queries to sample sites.

Site	Query	Number of results	XML file size, KB	Transformation time, ms
AutoTrader.com	2000-2004 Acura of any model within 25 miles from ZIP 00501	48	11	6376
	1983-2004 Ford Escort within 25 Miles from ZIP 10001	120	36	17657
	1983-2004 Jaguar of any model within 50 Miles from ZIP 00501	202	55	34141
	1983-2004 Land Rover Range Rover within 200 Miles from ZIP 00501	310	87	174863
	1990-2004 Cadillac with mileage under 75,000 within 50 Miles from ZIP10001	430	113	225767
	1991-2004 Toyota with price range from 10,000 to 15,000 within 50 Miles from ZIP 10001	472	139	264660
	1983-2004 Ford of any model within 25 Miles from ZIP 10001	500	152	289084
Amazon.com	Search for "gardenia"	21	5	3032
	Search for "snooker"	190	51	100455
	Search for "intranet"	431	112	204034
	Search for "dock"	599	130	423400
	Search for "dot"	1001	234	702333
ArchitectureWeek.com	Search for "chalet"	35	17	50994
	Search for "tall building"	311	159	324437
	Search for "column"	619	312	572231
	Search for "street"	780	394	730452
	Search for "environment"	1053	473	852432
NationJob.com	Computers/I.T./Telecommunications: Computer Operator	10	3	8094
	Computers/I.T./Telecommunications: Software Design/Project Management	121	45	64484
	Accounting/Finance/Insurance: Banking	693	203	454328
	Education/Teaching/Child Care	1478	567	773671
IMDb.com	User comments to "Arrival"	58	19	5034
	User comments to "Once Upon a Time in Mexico"	201	66	15443
	User comments to "Star Wars V"	746	257	244543
	User comments to "Harry Potter and the Sorcerer's Stone"	1212	485	384254
CiteSeer.org	Search for "hidden web"	38	16	70343
	Search for "google"	523	237	173799
	Search for "p2p"	754	390	262405
	Search for "web"	1000	561	400541

Table 5
Time needed to transform results of sample queries.

5.2.2 Extraction Accuracy

Since the goal of our HW-STALKER is to extract query results from the QURE-Pagelets and transform them to XML, we adopt *precision* and *recall* as our

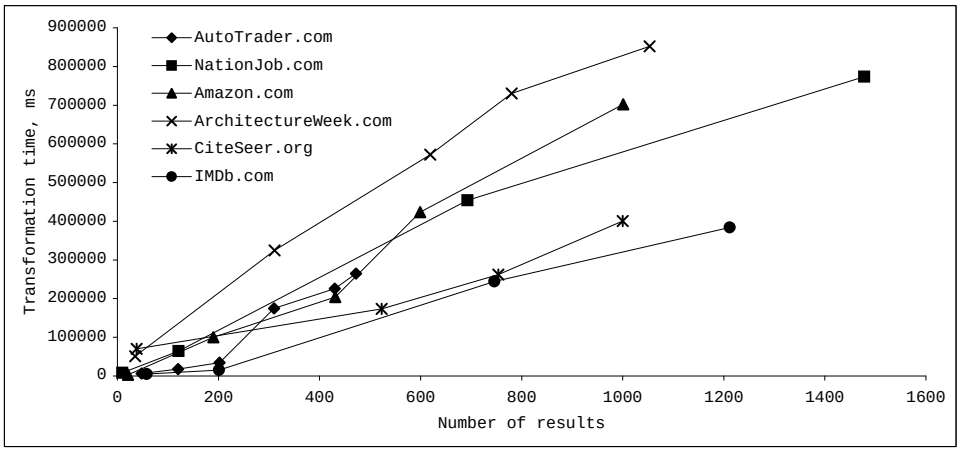


Fig. 19. Number of hidden web query results vs. extraction time.

Site	No. of Queries	No. of Query Results	No. of transformed XML Results	Recall (%)	Precision (%)
www.autotrader.com	7	2082	2468	81	96
Amazon.com	5	2242	2469	87	79
ArchitectureWeek.com	5	2798	2768	91	92
NationJob.com	4	2302	2452	98	92
IMDb.com	4	2217	1914	82	95
CiteSeer.org	4	2315	2240	90	93
RealEstate.yahoo.com	6	4004	4335	88	81
PubMed.com	9	3145	3191	100	98

Table 6
Precision and recall related to number of query results extracted.

Site	No. of Queries	No. of chain links in query results	No. of extracted chain links	Recall (%)	Precision (%)
www.autotrader.com	7	79	101	100	78
Amazon.com	5	221	193	84	96
ArchitectureWeek.com	5	277	274	91	92
RealEstate.yahoo.com	6	267	293	99	90
IMDb.com	4	109	98	89	99
CiteSeer.org	4	113	124	88	80

Table 7
Precision and recall related to chain link extraction.

performance measurement of extraction accuracy. For each set of query results, we manually extract the results and compare with the ones extracted by HW-STALKER. We measure the results in the following three ways.

Site	No. of Queries	No. of fanout links in query results	No. of extracted fanout links	Recall (%)	Precision (%)
PubMmed.org	9	3145	3171	100	99
Amazon.com	5	2242	2494	99	89
NationJob.com	4	2302	2423	100	95
RealEstate.yahoo.com	6	4004	4299	97	90
CiteSeer.org	4	2315	2239	88	91

Table 8
Precision and recall related to fanout link extraction.

- *Number of results returned:* Let Q_s denote the set of query results returned by the hidden web site. Let Q_h be the set of transformed results in XML format using HW-STALKER. Then the following formula P_q and R_q calculate the precision and recall respectively.

$$P_q = \frac{Q_s \cap Q_h}{Q_h}, \qquad R_q = \frac{Q_s \cap Q_h}{Q_s}$$

  Table 6 summarizes the results for the six hidden web sites.
- *Chain links extraction:* Let C_s denote the set of chain links in the query results returned by the hidden web site. Let C_h be the set of chain links extracted by HW-STALKER. Then the following formula P_c and R_c calculate the precision and recall respectively.

$$P_c = \frac{C_s \cap C_h}{C_h}, \qquad R_c = \frac{C_s \cap C_h}{C_s}$$

  Table 7 summarizes the results for the six hidden web sites.
- *Fanout links extraction:* Let F_s denote the set of fanout links in the query results returned by the hidden web site. Let F_h be the set of fanout links extracted by HW-STALKER. Then the following formula P_f and R_f calculate the precision and recall respectively.

$$P_f = \frac{F_s \cap F_h}{F_h}, \qquad R_f = \frac{F_s \cap F_h}{F_s}$$

  Table 8 summarizes the results for the six hidden web sites.

Note that we do not measure the accuracy of extraction of different attributes in a query result as such effort has already been reported in [11,20]. We only report accuracy of those features that are unique to HW-STALKER. Also, the results from this set of experiments show that we can achieve reasonably good performance across heterogeneous hidden web sites. Our results show relatively few items are extracted or transformed incorrectly. This is mainly due to the heterogeneous nature of HTML format. The server-side applications sometimes generate unexpected markup, provide incomplete information, or

Fig. 20. Query results containing 3 banners.

sometimes due to the existence of banners in unexpected places. For example, consider the Autotrader web site. Suppose we wish to search for `"toyota within 50 miles from zip 10001"`. The query returns around 500 results and most of these results contain at the most two banners. However, there are two results that contain 3 banners (Figure 20). In this case, HW-STALKER fails to perform correct extraction as it is only trained for case with 0 to 2 banners.

6 Related Work

The machine learning community has carried out research on learning extraction rules which occurred in mainly two contexts: creating wrappers for information agents and developing general purpose information extraction systems for natural language text. We review some of these technologies here. **WIEN** [15] takes as input a set of example pages where data of interest is labelled, and the output is a wrapper that is consistent with each labelled page. A specific induction heuristics is used to generate specific wrappers. The pages to be wrapped are assumed to have the same predefined structure as the examples. WIEN do not support nesting objects. In WIEN, items are expected to be always presented and ordered in the same manner. The work in [8] presents a language for wrapper development as part of TSIMMIS project. The main shortcoming of this work is that a user must examine the document and find the HTML tags that separate the objects of interest, and then write a program to separate the object regions. The whole process of discovering object boundaries is carried out manually . **SRV** [7] is an approach for wrapper generation based on the Natural Language Processing (NLP). SRV is a tool for learning extraction rules of text documents, based on a given set of training examples. It also relies on a set of token-oriented features that can be either simple or relational, thus it can be applied to extract information from HTML. SRV distinguishes a number of HTML-specific features related to HTML tags, e.g., *in-p* or *after-b*, This makes SRV able to extract data from HTML documents effectively. SRV is a single-slot tool, like WIEN, thus it supports neither nesting nor semistructured objects. A technique for supervised wrapper generation based on a logic-based declarative language called *Elog* is presented in [2]. This

technique is implemented in a system called Lixto which assists the user to semi-automatically create wrapper programs by providing a fully visual and interactive user interface. With Lixto, expressive visual wrapper generation is possible.

Compared to WIEN, which extracts all items at one time, in our approach we use several single slot rules based on STALKER. Also, we support nesting unlike WIEN and SRV. Compared to Lixto, our approach is much more user-friendly as it is based on STALKER. As user should define extraction rules himself in Lixto, it is a complicated task even using GUI. Another significant strength of STALKER compared to Lixto is that a user does not need to know HTML or to construct any extraction rules by himself. The only task for user is to construct $\mathcal{EC}$ description of the page that can be done without looking into a source code of the page and to mark several sample pieces of information in HTML using GUI. The flexibility and universality of STALKER extraction rules is based on usage of the wildcards, including domain-specified wildcards. Like Lixto extraction rules, STALKER extraction rules are iterative and designed to extract nested information. However, STALKER is able to construct wrappers only for single HTML pages but not for the sets of pages. Unlike the above approaches, HW-STALKER is developed specifically for hidden web data and hence is able to extract key attributes (identifiers and facilitators) from the query results. Our system is also focused on extracting data from dynamically generated hyperlinked web pages only.

RoadRunner [5] is the HTML-aware tool for automatic wrapper generation based on the inherent features of HTML documents. ROADRUNNER runs by comparing the HTML structure of two (or more) sample pages believed to have similar structures, and as a result, generates a schema for the data contained in the pages. From this schema, a grammar is inferred which is capable of recognizing instances of the attributes identified in the sample pages. The extraction process is based on an algorithm that compares the tag structure of the sample pages and generates regular expressions handling structural mismatches found between the two structures. The unique feature that distinguishes ROADRUNNER from all other approaches is that the process of wrapper generation is fully automatic and no user intervention is requested. However, such flexibility poses disadvantage as far as extracting hidden web data is concerned as it cannot extract identifiers and facilitators automatically.

Caverlee et al. [3] introduce the concept of a *QA-Pagelet* to refer to the content region in a dynamic page that contains query matches. Our notion of QURE-Pagelet is similar to this. The authors present a system called THOR that focus on discovering and extracting QA-Pagelets from the hidden Web. First, pages from each web site are grouped into clusters of structurally similar pages. Then, pages from the top-ranked clusters are examined through a subtree filtering algorithm. Our approach differs in the following ways. First,

we use machine learning-based mechanism to transform query results to XML format. Second, we identify the semantic constraints associated with the data values (using identifiers and facilitators) during the transformation process. THOR does not exploit such semantic constraints. Such semantic constraints are useful in several applications such as change detection, hidden web data integration etc.

7 Conclusions

In this paper, we present a machine learning-based approach for extracting relevant hidden web query results and transforming them to XML. We propose an algorithm called *HW-Transform* for transforming QURE-pagelets to XML format. In our approach, we extend the STALKER technique to extract results from a set of dynamically generated hyperlinked web pages. The XML representation of query results encapsulates only the data that a user is interested in. One of the key features of our approach is that a user maps special *key attributes* in query results, called *identifiers* and *facilitators*, into XML attributes. These attributes facilitate change detection, data integration etc. by efficiently identifying related results. As a proof of concept, we have implemented a prototype system called HW-STALKER using Java. Our experiments demonstrate that *HW-Transform* shows acceptable performance for transforming QURE-pagelets to XML.

References

[1] Z. Bar-Yossef and S. Rajagopalan. Template Detection via Data Mining and its Applications. In *Proceedings of the World Wide Web Conference*, 2002.

[2] R. Baumgartner, S. Flesca, and G. Gottlob. Visual Web Information Extraction with Lixto. In *Proceedings of the 27th VLDB Conference*, Roma, Italy, 2001.

[3] J. Caverlee, L. Liu, and D. Buttler. Probe, Cluster, and Discover: Focus Extraction of QA-Pagelets from the Deep Web. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2004.

[4] S. Chakrabarti, M. van den Berg, and B. Dom. Focused Crawling: A New Approach to Topic-Specific Web Resource Discovery. In *8th World Wide Web Conference*, May 1999.

[5] V. Crescenzi, G. Mecca, and P. Merialdo. RoadRunner: Towards Automatic Data Extraction from Large Web Sites. In *Proceedings of the 26th International Conference on Very Large Database Systems*, pages 109–118, Roma, Italy, 2001.

[6] M. Diligenti, F. Coetzee, S. Lawrence, C. L. Giles, and M. Gori. Focused Crawling using Context Graphs. In *26th International Conference on Very Large Databases, VLDB 2000*, September 2000.

[7] D. Freitag. Machine Learning for Information Extraction in Informal Domains. *Machine Learning*, 39, 2/3:169–202, 2000.

[8] J. Hammer, H. Garcia-Molina, S.Nesterov, R. Yerneni, M. Breunig, and V. Vassalos. Template-Based Wrappers in the TSIMMIS System. *SIGMOD Record*, 26, 2:532–535, 1997.

[9] H.Davulku, J.Freire, M.Kifer, and I.V.Ramakrishnan. A Layered Architecture for Querying Dynamic Web Content. In *ACM Conference on Management of Data (SIGMOD)*, June 1999.

[10] M. K.Bergman. The Deep Web: Surfacing Hidden Value, September 2001. http://www.brightplanet.com/deepcontent/tutorials/DeepWeb/deepwebwhitepaper.pdf.

[11] C. A. Knoblock, K. Lerman, S. Minton, and I. Muslea. Accurately and Reliably Extracting Data from the Web: A Machine Learning Approach. *IEEE Data Engineering Bulletin*, 23(4):33–41, 2000.

[12] D. Konopnicki and O. Shmueli. Information Gathering in the World-Wide Web: The W3QL Query Language and the W3QS System. *ACM Transactions on Database Systems*, 23(4):369–410, 1998.

[13] V. Kovalev. Change detection to the hidden web. Master's thesis, School of Computer Engineering, Nanyang Technological University (Singapore), 2003.

[14] V. Kovalev, S. S. Bhowmick, and S. Madria. HW-STALKER: A Machine Learning-based Approach to Transform Hidden Web Data to XML. In *Proceedings of the 15th International Conference on Database and Expert Systems Applications (DEXA 2004)*, 2004.

[15] N. Kushmerick. Wrapper Induction: Efficiency and Expressiveness. *Artificial Intelligence Journal*, 118, 1-2:15–68, 2000.

[16] S. Lawrence and C. L. Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, April 1998.

[17] S. Lawrence and C. L. Giles. Accessibility of Information on the Web. *Nature*, 400:107–109, July 1999.

[18] A. McCallum, K. Nigam, J. Rennie, and K. Seymore. Building Domain-specific Search Engines with Machine Learning Techniques. In *Proc. AAAI-99 Spring Symposium on Intelligent Agents in Cyberspace*, 1999.

[19] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, and G. Sindoni. The ARANEUS Web-base Management System. In *Proceedings of the International Conference on Management of Data*, pages 544–546, 1998.

[20] I. Muslea, S. Minton, and C. A. Knoblock. Hierarchical Wrapper Induction for Semistructured Information Sources. *Autonomous Agents and Multi-Agent Systems*, 4(1/2):93–114, 2001.

[21] D. Shestakov, S. S. Bhowmick, and E.-P. Lim. DEQUE: Querying the Deep Web. *Data and Knowledge Engineering Journal*, 52(3):273–311, 2005.