In this paper we focus on the latter case through building a robust system to collect structural data from WWW continuously. It is a part of a collaborative project between Renmin University of China and City University of Hong Kong, the goal of which is to build a fully-fledged multimedia recipe database by collecting as many recipe web pages as possible. We extract the data records from the collected recipe pages which will be later on used in a multimedia database application–*RecipeView* (Fig.2). Generally speaking, recipe web pages are very similar to online product web pages in that (a) one web page contains only one record, (b) they follow an underlying template, and (c) there are many optional attributes. Some examples of recipe pages are shown in Fig.1. Thus by applying existing techniques, which are roughly classified into two categories– wrapper induction and automatic extraction, our goal may be achieved. However, this turns out to be a non-trivial task because of the following reasons:

– It is unpractical to crawl all the recipe pages from a web site. In Fig.1(c), there is an example of a recipe category list. The webmaster will add/update some new recipe links (shown in red circle) while updating other links such as advertisements and activities. Naive crawling of all updated links will not only lead to an inefficient strategy but also impact the latter steps by introducing some noisy web pages. Thus we have to consider how to identify real recipe links while crawling pages incrementally.
– It is almost impossible to induce a general wrapper with initial batch of recipe web pages. Because of the continuous updating of recipe web sites, the changes of the underlying schema may cause the existing wrapper broken. For example, Fig.1(a) is a typical recipe web page when the web site was created. It only contains a name, a picture, a material list, a seasoning list and some cooking steps. As time elapses, the webmaster provides us with some new recipes, one of which is shown in Fig.1(b). Because some complex new optional attributes are added (e.g. two styles of sauce in Fig.1(b)) and the existing attributes are revised (e.g. seasoning turns to be repeatable), such of these variations not only cover simple representation changes, but also involve serious schema evolutions, which definitely makes conventional extraction techniques inapplicable.

Due to these observations, our approach is to build a system (called *RecipeCrawler*) that can automatically extract relevant content data, and be able to do so incrementally so that the new web pages containing new recipe records may be added dynamically. To this end it must support the following incremental features in extraction of newly crawled web pages from the recipe websites.

1. **Incrementally crawling specific web pages**. In our system, some web data sources, such as recipe web site's categories, recipe blog pages, or even recipe online forums, are monitored. Whenever the links are updated, crawler should not only grab the web page pointed by the link, but also justify whether it is the one we need. It is possible as we have some extracted recipe data records, which can give us the domain knowledge of recipes.
2. **Incrementally extracting web pages for data records**. Either wrapper based or automated method faces the problem of web site's schema evolutions. The extraction program should not only be able to adapt itself to meet the schema revision,

Fig. 1. Examples of Recipe and Category List Web Pages

but also be able to identify new attributes. This is important to help applications which rely on the extraction system to be of more concrete, useful, and valuable services. And it also enables the extraction system to be a reasonable and practical web data extraction system.

By putting all things together, we aim to build our system as a practical robust system which supports incremental automated data extraction. It is different from existing systems in that novel modifications are made upon the tradition architecture. In a nutshell, our contributions in this paper include: 1) a framework for building incremental web data extraction system, which is implemented in our prototype system for collecting recipe data from WWW incrementally; 2) solutions for adopting and adapting existing data extraction techniques under incremental scenarios.

In this paper we describe our *RecipeCrawler* system in detail. The rest of this paper is organized as follows. In section 2, we briefly review some existing techniques on web data extraction. Section 3 gives out an overview of *RecipeCrawler*. Section 4 and 5 discuss our main considerations in designing and implementing each component. Finally we give out a conclusion and future works in section 6.

2 Relate Work

One of the reasons why the Web has achieved its current huge volume of data is that a great and increasing number of data-rich web sites automatically generate web pages according to the data records stored in their databases. Taking advantage of this fact, several approaches have been proposed and systems have been built to extract these data in literature. Generally these systems fall into two categories: wrapper induction versus automatic extraction.

With wrapper induction techniques, some positive web pages are selected as positive examples and then wrappers are trained. Though using wrappers to do continuous extraction is possible, wrappers may expire in future [6]. Thus wrapper maintenance problems arose and efforts were paid in solving it. However, to our knowledge, it assumes that there are only few small changes in web pages' representation whereas in fact the underlying schema may change [8], such as:(1) attributes that have never appeared in previously extracted pages may subsequently be added; (2) attributes appeared in previously extracted web pages may later be removed. These can cause the templates induced from existing web pages to be invalid, thus intuitive extraction strategies can not be applied. Therefore wrapper induction is not practical towards long-time, continuous data extraction.

On the other hand, as automatic extraction techniques can automatically extract structural data without doing wrapper maintenance from web pages, it becomes more popular recently years. The first reported work on automated data extraction was done by Grumbach and Mecca [5], in which the existence of collections of data-rich pages bearing a similar structure (or schema) was assumed. In RoadRunner [3], an algorithm was proposed to infer union-free regular expressions that represent page templates. Unfortunately, this algorithm has an exponential time complexity hence it is impractical for real-life data extraction systems. Then Arvind and Hector [1] proposed an improved version with a polynomial time complexity by employing several heuristics. Both of these works view web pages in HTML as a sequence of tokens (single words and tags), so when it comes to infer a template from complex web pages with many nesting structures, their solutions are still inapplicable. Other researchers have tried to solve the automated data extraction problem by viewing web pages as a long string, through employing similar generalization mechanisms (e.g., [2] and [10]). Be aware of the tree structure of web pages, [9] and [11] presented techniques based on tree edit distance for this problem. Both of them utilize a restricted tree edit distance computation process to find mapping between two web pages and then do future data extraction. In [9], wildcards are attached to tree nodes and heuristics are employed when there is a need to generalize them. In [11], a more advanced technique named partial tree alignment was proposed, which can align corresponding data fields in data records without doing wildcards generalizations. In our system, we use a similar technique and make it applicable under incremental data extraction.

While some major works have been done on clustering or classifying web pages, few of them are on automated data extraction as far as we can see from the literature. In [4], several web page features were proposed for wrapper-oriented classification. In the news extraction system [9], a hierarchical clustering technique was proposed to cluster web pages according to their HTML tree structures. A basic distance measure-edit distance is calculated by comparing two HTML DOM trees, which can tell us how similar the two web pages are. When the web page contains more than one data record, there is almost no need to do the clustering. But new problems do arise. For example, how to identify data regions containing data records in such kind of web pages is a problem. In particular, several strategies have been proposed in [7] and [12].

Combining these existing automated data extraction techniques may lead us to a generic system that is able to crawl, cluster and extract structured data from a whole

web site once for all. For our recipe collection scenarios, we need to continuously collect recipe data from the web, hence modifications to such techniques or other novel techniques are needed. In the rest of this paper we show our approach to build an incremental data extraction system by adopting and adapting the existing web data extraction techniques.

3 *RecipeCrawler* - a Recipe Data Collection System

Starting from this section, we will discuss the general considerations on how to build a system to support incremental features in conventional architecture by introducing our recipe data collection system. As Fig.2 illustrates, general architecture of current existing extraction systems were applied. Besides adopting and adapting the classic components such as web crawler, web data extractor and annotator, a new component called "*Monitor*" is advocated to keep a close watch on recipe sources. Instead of digging into the details on how it is designed and implemented as well as how it supports incremental features, in this section we would like to give an overview on how recipe data are collected.

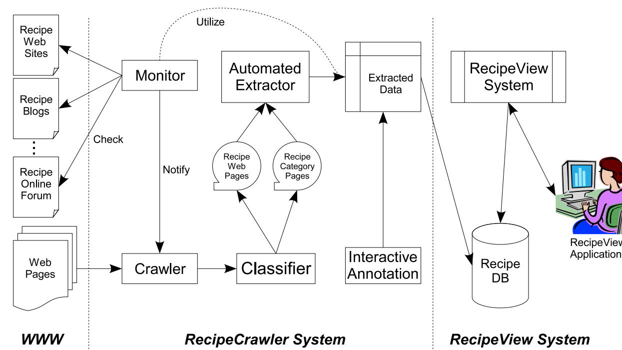

Fig. 2. Recipe Data Extraction System - An Overview

The mission of *RecipeCrawler* is to provide *RecipeView* with the recipe data records which are embedded in web pages. Here *RecipeView* is a user-centered multimedia view application built on top of the recipe database and means to provide user continuous, flexible user experience. It requires the extraction system (viz. *RecipeCrawler*) to be incremental because it needs recipe data updated every day on WWW.

Fig.2 shows an overall picture on how *RecipeCrawler* works. In particular, we incrementally grab recipe web pages by monitoring some data sources, which are as shown in the left part of Fig.2, including recipe web sites, recipe blogs and recipe online forums et. al. Considering that their indices are usually accessible (such as category lists in recipe web sites, taxonomy pages in recipe blogs and archive lists in recipe online forums), we establish a module called "*Monitor*" to find out the updated links from these sources. In order to identify whether the specific updated link is just the one we need,

extracted data has been used as domain knowledge to do data clarifications. And survivors, which are definitely the ones we need, are sent to "*Crawler*" which does basic crawling as well as validation and repairing on HTML pages.

Next the crawled web pages are delivered to the "*Classifier*" which puts pages into different categories. In this procedure, an algorithm proposed in [9] has been adopted and adapted to classify web pages according to their underlying structures (or underlying template). Two categories–"*Recipe Web Pages*" and "*Recipe Category Pages*" are derived through this step, where the former one usually contains the detailed information of each recipes and the latter one usually maintains taxonomy of recipes.

In the extraction procedure, web pages in each category are processed by an "*Automated Extractor*" and thus category information and recipe data are retrieved. Annotation was done by a module named "*Interactive Annotation*" which is operated by human, who tells the system what attribute is about what. As our system means to work in incremental way, being able to handle schema changes is critical so we proposed a method by adopting algorithms in [11]. We will further discuss it in section 5 as well as the mechanism of annotation process. So finally we get the desired data with corresponding annotations and thus can import them into DBMS for future applications, which is *RecipeView* in our case.

Before we go to the sections that discuss the details of each component, we want to emphasize the incremental nature of *RecipeCrawler* again. Incremental features in *RecipeCrawler* are the basic requirements and also the significant differences comparing with other systems. Though there is an initial web page set, which can be extracted before the *RecipeView* system is established, we can not guarantee that the wrapper induced or the schema learnt in them will always be valid for future cases, because we can not naively believe the webmaster will always update recipes activities, or assume the schema will not change. In other words, our *RecipeCrawler* should face the very dynamic perspective of WWW and the only choice is to make sure that each component of our system has the ability of doing incremental update.

4 Retrieving Recipe Web Pages

Monitoring, crawling and classifying procedures in *RecipeCrawler* are implemented to retrieve recipe web pages. In this section we mainly focus on the mechanism of monitoring and classifying procedures whereas crawling procedure is omitted because its implementation is fairly simple and straightforward.

4.1 Monitoring Recipe Data Sources

Recipe data sources on WWW usually have an index facility, such as category lists in recipe web sites, taxonomy pages in recipe blogs and archive lists in recipe online forums and so on. Monitoring them for updated recipe links generally should (1) find out whatever new/updated links, and (2) identify whether they are recipe-related links or not. The former step is easy by simply comparing current web page with history version whereas the latter one is complicated. The link discovery procedure of conventional crawler usually does simple identifications based on several rules, such as URL

domains, file types and so on. Few works are done on semantic link discovery because: (1) crawlers are usually of general use; (2) insufficient domain knowledge can be utilized to do it. However, in *RecipeCrawler*, we focus on recipe web pages, concerning not to introduce noisy web pages to subsequent procedures; we can even have domain knowledge by analyzing the extracted data of the initial set, which can always be selected out when first time we crawl the web site. With these characteristics in mind, we proceed to present a semantic link discovery method.

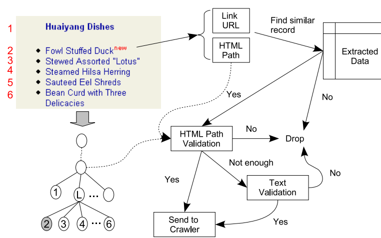

Fig. 3. Identifying Recipe Links Based on the Extracted Data - An Example

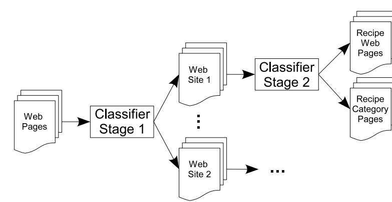

Fig. 4. Classifying Recipe Web Pages

As illustrated in Fig.3, our strategy of identifying recipe links on the basis of the extracted data works as follows. First, the current index of a web page is compared to the old one. In this way, the updated links, texts and HTML paths can be retrieved. For example, “Fowl Staffed Duck”(in short, FSD) with its link and HTML DOM path can be retrieved. Secondly we try to find records in extracted data which have similar links, as machine does not know whether it is a recipe link. Two links are similar if we can find a common pattern in them (In our system, we uses common URL prefix). Only considering URL pattern is sometimes not enough as there are still some links such as activities may survive. Therefore we utilize HTML paths and texts for further clarifications. After finding out similar records of a specified link, we first check how many records residing in the same subtree according to HTML paths. Referring back to the example, as we have FSD’s DOM path, we also know similar records’ DOM path (which are recorded in last time’s extraction), by finding common parent nodes, such as “L” node of the DOM tree in right bottom corner of Fig.3. Note that we only give a simple DOM tree here due to the space reason, in which number denotes the content. If we can not find any, this link is probably not a recipe link so we discard it. If we can only find few (in our system, we use 0.5 as the threshold, which means half out of total records), the text is used as the third judgment, which is simple keyword matching in our system, in the hope of finding common recipe keywords(such as “Beef”, “Pork” and so on). If most records reside in the same subtree, we let the link survive. Figure 3 illustrates the whole process we have just described, which, based on our practice, has been quite effective and efficient.

4.2 Classifying Recipe Web Pages

In the next step, we build a module "*Classifier*" to handle the web page classification. The classifier program in our system has two stages, as shown in Fig.4. In the first stage we organize the web pages according to URLs, thus obtaining categories of web sites. This stage is relatively easy. Next we further classify crawled web pages according to the tree structures. A clustering algorithm based on tree edit distance [9] has been adopted and adapted. As mentioned before, recipe web pages in our scenario may contain repeatable attributes, so we have modified the matching process to cover repeatable cases. It is called sibling matching which is also used in automated extraction procedure and the details will be given in section 5.1. After classification we will get two categories, namely recipe web pages and recipe category pages, for each web site. Subsequent extractions will be done in these categories.

The classification procedure is in nature incremental for cases where there are no big changes in page templates. When a template (or structure) changes a lot, a new initial data set needs to be generated so that a new classification process can proceed.

5 Retrieving Recipe Records

We now describe how *RecipeCrawler* retrieves recipe data from the crawled recipe web pages. There are two modules involved, namely "*Automated Extrator*" and "*Interactive Annotation*". Though they do different functions in retrieving recipe data, there is no rigid execution order. In *RecipeCrawler*, they are actually invoked asynchronously. Fig.5 gives an illustration on how these two modules cooperate with each other. The *Automated Extractor* continuously does extraction on web pages while the *Interactive Annotation* is notified each time new attributes are identified. *Automated Extrator* will generate two data tables, namely "*Recipe Data*" and "*Category Data*", from recipe web pages and recipe category pages, respectively. Each table may contain some new attributes during the incremental extraction. Thus an execution of annotation procedure is needed. Then we select data fields that have been annotated from these two tables, and join them according to URLs. Finally data is extracted and ready to be imported into DBMS.

5.1 Automated Extraction

In this module, we adopt techniques proposed in [11] for automated extraction. As reported in [11], an algorithm named *partial tree alignment* based on the simple tree matching was used to extract data records in data intensive web pages, such as result pages returned by online retailer web sites. The recipe category web pages in our system are also data intensive web pages, so data records can be directly extracted by applying this algorithm. But we need to modify it in order to extract new/updated records in it for supporting incremental features. This can be done by comparing currently extracted results to the former ones, so the details are omitted here.

On the other hand, extracting data from recipe web pages is not so easy. It is a non-trivial problem because: (1) attributes that have never appeared in previously extracted

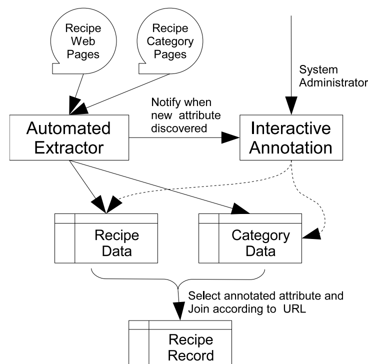

Fig. 5. Retrieving Recipe Data from Web Pages

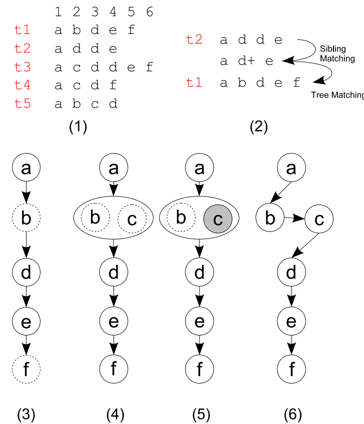

Fig. 6. Illustration of How Automated Recipe Data Extraction Works

pages may subsequently be added; (2) attributes that appeared in previously extracted web pages may later be removed; (3) attributes that appeared as singleton in previously extracted web pages may be modified to be repeatable. For example, referring back to Fig.1, the “sauce” attribute appearing in Fig.1(b) is an example of added attributes, and the “seasoning” attribute appearing both in Fig.1(a) and Fig.1(b) is an example of revised attributes, which later can be repeatable. There is no example of removed attributes in Fig.1, but it is easy to give out: any optional attribute can be it when we start from web pages containing it to web pages without it. Though the technique proposed in [11] can roughly handle these situations by selecting and starting from the maximal web page in the hope of that it contains as many optional nodes as possible, it is unfortunately inapplicable in our incremental crawling scenario. So we have adapted it to fulfill the incremental requirements.

Instead of explaining the detailed algorithm used by *RecipeCrawler*, we give an illustrative example in Fig.6 to show how it works. We suppose there are 5 recipe web pages, and to be simple, we present them in simple characters sequence, in which each character denoting a subtree directly contains text values, such as “<LI>Materials:
Beef 150g</LI>” . We can get the sequence by specific traversal of HTML tree [11], and we use pre-order traversal here. According to [11], partial tree alignment first selects the biggest web page as the seed and then do multiple tree alignment. In our example, the biggest one is t3. But in an incremental situation, t3 may not be in the initial set because it is not created by any webmaster at all. In our example, we assume that the initial set has t1 t2, whereas t3, t4 and t5 are added subsequently.

For the initial set we apply the partial tree alignment technique. First we do a sibling matching (as shown in Fig.6(2)), which is used to handle repeatable attributes (“d” in t2). The sibling matching scans each tree and tries to match siblings in it. If two sibling nodes match, they will be replaced by a single example node (we simply take the first one). We do not consider non-sibling nodes because usually a list of repeatable

attributes will not be interrupted by other attributes. (For example, the webmaster will not insert some cooking steps in the middle of listing materials.) And the sibling matching performs whenever we match a web page to another (as well as the template, see below). After doing that we make the tree matching based on the edit distance computation to find mappings. By taking the biggest one t2 as the template, we can align t1 to it and by applying partial tree alignment techniques [11] we can also align optional nodes. The basic idea of partial template alignment is trying to find the unique insertion location for each unmatched nodes. In our example, "b" of t1 is unmatched, but we can find a unique insertion location in t2 for it, because "a" and "d" are matched and there is nothing between them in t2. So "b" should be inserted between "a" and "d" in t2 to form a template. After inserting all optional nodes as proven in [11], recipe data is extracted and a template (shown in Figure 6(3)) is induced. Then an annotation process may be invoked, but at this time we are not sure that the nodes "b" and "f" are the data attributes we need (they can be useless values such as "copyright by" et. al.). Another reason is that they may be disjunctions as we have only few instances. So, in our example, we simply suppose no annotation in it, so actually we only extract "a", "d" and "e".

Now we come to the part of incremental extraction. Supposing that t3, t4 and t5 will be updated and crawled one by one, Fig.6(4,5,6) shows how the extraction is done. The basic idea is to match new crawled web pages with the existing template and insert the unmatched nodes into the template. When there is no unique insertion location for the specific node, we insert it by merging it as a possible value into a possible node. In our example, when t3 comes, we find that "c" does not have a unique insertion location as there is already an unannotated "b" between "a" and "d", so we merge "c" as a possible value into "d", thus the template can cover t3 (as shown in Fig.6(4)). At this time t3 can be partially extracted with some part left in the induced template, which may be further matched or annotated (extraction process will give annotation process a notification at this time). Another node, say "f", matches with the one in the template, thus we have enough instances to identify "f" as an attribute and both "f" nodes in t2 and t3 will be extracted.

After processing t3, t4 comes in subsequently. This time we match it with the template too. The difference is that when matching with node "b c", we need to match two times to find the best one. We can see that "c" will be matched thus attribute "c" will be identified. But we can not take it out from the "b c" node for there is still no unique insertion location. The template after matching and extracting t4 is as be seen in Fig.6(5). After t5 comes, matching with t5 will identify the attribute "b" too. And the order of attributes "b" and "c" can be identified since we have t5 as the instance (there is a "b" "c" sequence in t5). Thus all attributes are identified and can be extracted. The induced template is shown in Fig.6(6). Next time when new web pages come in, the same processing techniques can be used.

Note that currently we do not consider disjunctions in our strategy due to two reasons. Firstly, disjunctions are actually not that serious when we are doing incremental extraction. By using following web pages as examples (Fig.6(6)), identifying whether there are disjunctions is easy. Secondly, the chances of disjunctions making our strategy broken are fairly few. For example, considering a web page t6("a c b d e"), our strategy would break while handling it. But this is rare because t6 means that web master

changes the order of attributes (such as giving “cooking steps” before “materials”). It is almost unlikely and we did not find any cases in our practice, so we leave this problem to be a possible future work.

5.2 Interactive Annotation

Currently in *RecipeCrawler* the annotation procedure is designed as an interactive program. It can be asynchronously invoked by a system operator while the system does automated extraction. The template induced by automated extraction will be presented to the operator for annotation instead of requiring him to do annotation on each record. When a new attribute is identified, a notification will be given. Then the system operator can check the revised template and examples to decide what kind of attribute it is. Having annotations made to the extracted recipes and category data, they will be selected out and joinned based on URLs to generate the final extraction results. Unannotated data will be reserved in the extracted data storage for future annotation. This mechanism ensures us to be able to incrementally extract meaningful recipe data for *RecipeView* as soon as newly crawled web pages come in. In our practice, we perform the interactive annotation when the initial set was extracted and when enough (e.g., 10) new web pages are extracted. The current practice of *RecipeCrawler* shows that such an approach is quite reasonable and effective.

5.3 Importing Recipe Data Records

As shown in Fig.2, the extracted recipe data records by *RecipeCrawler* are to be utilized by a front-end application system called *RecipeView*. Since the retrieved recipe data records come from various sources, they should go through an importation procedure before they can be fully utilized. This procedure is called “Preprocess” in *RecipeView*, which involves *Filtering* and *Standardization*. The *Filtering* module makes sure that all the recipe records are qualified for the system requirements (e.g. by checking whether the data fields of each record are correctly identified). In the *Standardization* module, all the recipe records have to conform to a standard presentation by fusing different data formats together. For instance, the display sequence of the data fields in each record must be the same. Thus they become uniform and consistent. After the “Preprocess” procedure, the recipe data records are imported into an underlying DBMS for possible user manipulations within the *RecipeView* system.

6 Conclusion and Future Work

As we believe, building incremental data extraction is a critical step towards practical, continuous, reliable web data extraction systems that utilize WWW as the data source for various database applications. In this paper, we have described such a system (viz., *RecipeCrawler*) which targets at incrementally collecting recipe data from WWW. General issues in establishing an incremental data extraction system are considered and techniques applied to recipe data collection from the Web. Our *RecipeCrawler* has served as the backend of a multimedia database application system (called *RecipeView*)

and offers good experimental results. Various techniques proposed in literature for data extraction from WWW are adopted and adapted to do the automated recipe data extraction as well as to support incremental features. As for future research, besides evaluating and improving our system, we also plan to address other important issues, including better crawling strategies and automated annotation algorithms.

7 Acknowledgments

This research was partially supported by the grants from the Natural Science Foundation of China under grant number 60573091, 60273018; the National 973 Basic Research Program of China under Grant No.2003CB317000 and No.2003CB317006; the Key Project of Ministry of Education of China under Grant No.03044 ; Program for New Century Excellent Talents in University(NCET).

References

1. A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *Proceedings of the* 22th *ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2003.
2. C.H. Chang and S.C. Lui. Iepad: information extraction based on pattern discovery. In *Proceedings of the* $10th$ *International World Wide Web Conference*, pages 681–688, 2001.
3. V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of* 27th *International Conference on Very Large Data Bases*, pages 109–118, 2001.
4. V. Crescenzi, G. Mecca, and P. Merialdo. Wrapping-oriented classification of web pages. In *Proceedings of the* 17th *ACM Symposium on Applied Computing (SAC)*, pages 1108–1112, 2002.
5. S. Grumbach and G. Mecca. In search of the lost schema. In *ICDT '99*, pages 314–331, 1999.
6. N. Kushmerick. Wrapper verification. *World Wide Web*, 3(2):79–94, 2000.
7. B. Liu, R. L. Grossman, and Yanhong Zhai. Mining data records in web pages. In *Proceedings of the* 9th *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 601–606, 2003.
8. X. Meng, D. Hu, and C. Li. Schema-guided wrapper maintenance for web-data extraction. In *the* 5th *ACM CIKM International Workshop on Web Information and Data Management*, pages 1–8, 2003.
9. D.C. Reis, P.B. Golgher, A.S. Silva, and A.H.F. Laender. Automatic web news extraction using tree edit distance. In *Proceedings of the* 13th *international conference on World Wide Web*, pages 502–511, 2004.
10. J. Wang and F. H. Lochovsky. Data extraction and label assignment for web databases. In *Proceedings of the* 12th *International World Wide Web Conference*, pages 187–196, 2003.
11. Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *Proceedings of the* 14th *international conference on World Wide Web*, pages 76–85, 2005.
12. H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. T. Yu. Fully automatic wrapper generation for search engines. In *Proceedings of the* 14th *international conference on World Wide Web*, pages 66–75, 2005.