The eShopmonitor: A comprehensive data extraction tool for monitoring Web sites

N. Agrawal
R. Ananthanarayanan
R. Gupta
S. Joshi
R. Krishnapuram
S. Negi

Typical commercial Web sites publish information from multiple back-end data sources; these data sources are also updated very frequently. Given the size of most commercial sites today, it becomes essential to have an automated means of checking for correctness and consistency of data. The eShopmonitor allows users to specify items of interest to be tracked, monitors these items on the Web pages, and reports on any changes observed. Our solution comprises a crawler, a miner, a reporter, and a user component that work together to achieve the above functionality. The miner learns to locate the items of interest on a class of pages based on just one sample supplied by the user, via the user interface (UI) provided. The learning algorithm is based on the XPaths of the Document Object Model (DOM) of the page.

1. Introduction

Reliability, timeliness, and correctness of information on Web sites are matters of concern to every system administrator. In the case of commercial sites, inaccuracies and factually incorrect information on the Web site could lead to serious losses and legal problems, apart from losing customer interest and goodwill. For example, an airline website could erroneously offer air tickets for unusually low prices, or an online store might display products with incorrect prices.[1] Further, the data displayed on Web pages might be derived from multiple dynamic upstream data sources, and errors could creep in because of some obscure error at the database end. Even when the pages are statically generated, there could be changes to the links of the page, leading to problems of missing and inconsistent links. While different kinds of checks may be enforced at the database level to ensure consistency and correctness of data, these are not always sufficient to trap all errors that arise on the Web pages. For instance, missing links, incorrect links, and incorrect or missing images are some of the problems that may arise on the site even when the database has been checked for consistency and correctness.

In view of these problems, it would be useful for the site owner or administrator to have a tool to monitor important Web pages and detect anomalies of the kinds mentioned above. To make this notion more concrete, let us define a *field or item of interest* as an HTML element whose value is of high importance to the Web site. For example, HTML elements that contain a product name, product price, promotion, or discount offer are all fields of interest. Therefore, a field of interest is any HTML element whose value should not display anomalous behavior. Since the values of these fields of interest must be extracted or "mined" from the Web page, we call the anomalies concerning these fields of interest *mining anomalies*. Other anomalies are called *crawling anomalies*, since they can be detected by a crawler. To summarize, we are interested in detecting the following kinds of anomalies:

1. Mining-related anomalies; for example,
   - Identify pages where some specific item of information has changed, e.g., pages where 'ProductPrice' has changed by more than 10%.

[1] Such events have actually taken place; consequently, many transactions have had to be reversed.

0018-8646/04/$5.00 © 2004 IBM

- Identify pages where the value of some specific item of interest is *x*, e.g., pages where 'ProductAvailability' equals 'Not In Stock'. Instead of =, other operations such as ≥, ≤, and *substring* can also be used.
- Identify pages where a particular field of interest is missing, e.g., pages where 'ProductImage' is missing.

2. Crawling-related anomalies; for example,
   - Navigational anomalies; e.g., there is no click-path between the laptop sub-site and the CD-ROM drive sub-site.
   - Missing or broken links.
   - Documents returning HTTP 404 (Page not found) or HTTP 50× (internal server errors), etc.

Anomalies can also be composed of more than one field of interest—for example, 'ProductName' = 'Desktop' and 'ProductPrice' ≤ $100. Generalizing this concept, an anomaly can be considered as a query with a set of constraints connected by logical operators. On execution, the queries may retrieve some results, which can then be classified as anomalies or non-anomalies by the user.

From the usage point of view, note that such a tool is of interest to the following kinds of users:

1. The Webmaster, who wants to eliminate broken links, internal server errors, etc.
2. The content manager, who wants to ensure that a) correct information is displayed, b) there is no missing information, c) there is no mismatching of information, and d) there are no anomalous changes in any field of interest.
3. The marketing researcher, who can use such a tool to study a competitor's Web site. For example, the laptop promotions offered by a competitor over the last month or so can be followed.

We have built the eShopmonitor to perform these tasks in a comprehensive end-to-end manner. The solution comprises three major components—a crawler, which retrieves pages of interest to the user; a miner, which allows the user to specify the fields of interest in the different kinds of pages and subsequently extracts these fields from the crawled pages; and a reporter, which generates reports on the gathered information. Further, the crawled data can be compared with snapshots of previously crawled data in order to detect anomalies and/or interesting changes in the fields of interest. To allow this, the eShopmonitor stores the last 30 crawls.

The initial version of the eShopmonitor was built to monitor the *ibm.com* Web site; this version has subsequently been extended to work on other sites.

In Section 2, we discuss related work in this area, i.e., information extraction from Web pages. Section 3 discusses the top-level architecture of the eShopmonitor and its different components at the functionality level.

In Sections 4, 5, and 6, respectively, we discuss the crawler, the miner, and the reporter in detail, highlighting the technology and design for each of them. We conclude by discussing some of the possible extensions of the eShopmonitor and future work.

2. Related work

Information extraction from semi-structured documents such as Web pages has concentrated primarily on the generation of wrappers for the pages. A *wrapper* may be defined as a software module that converts information on the HTML pages to a structured representation, closer to a relational database format, which can then be used for further processing. Reference [1] surveys some of the core technologies for adaptive information extraction, using machine learning, and classifies the basic techniques into two main categories: finite state approaches, which learn extraction knowledge that is equivalent to finite state automata (FSA); and relational approaches, which learn extraction knowledge essentially in the form of first-order Prolog-like extraction rules. It is noted that while the FSA approaches are simpler, with faster learning algorithms, the relational approaches are more expressive. Reference [2] identifies a family of six wrapper classes that wrap up to 70% of the sites surveyed, with some of the algorithms growing exponentially in the number of attributes and some requiring more training examples to converge than others. Reference [3] presents a set of tools for wrapper generation, using machine learning techniques which have been applied to detect the changes on Web pages and filter them using semantic concepts such as name, phone number, and other such properties. The system also responds to changes in the page layout and format by repairing the wrappers. Jedi [4] is a lightweight tool for the creation of wrappers and mediators to extract, combine, and reconcile information from several independent information sources. It provides a fault-tolerant parser based on ambiguous context-free grammars (CFGs) for the wrapper generation. Roadrunner [5] also investigates techniques for extracting data from HTML sites by automatically generating wrappers. By comparing two HTML pages at a time, it attempts to learn the structural patterns on the page by deducing the source data set from which the pages have been generated. References [6] and [7] are other works that discuss techniques for data extraction from the Web. IEPAD [8] is a system that automatically discovers extraction rules from Web pages using a data structure called *PAT trees* to discover patterns to mine on the Web and then repeatedly mines for these patterns.

The techniques discussed thus far have the advantage of providing a high degree of automation with little or no

manual intervention required. However, in typical commercial applications, higher levels of accuracy are required in order to use the tools for information extraction. Further, users wish to be able to point and click at items of interest that must be extracted, in a simple user-friendly manner. Users also wish to be able to associate different fields of interest as the pattern of interest to mine, regardless of the source page format or the patterns that are automatically mined. The wrapper generation techniques discussed thus far do not provide this level of flexibility and the level of accuracy required in commercial applications. Also, no visual support is provided for the end user.

Many information-extraction techniques have been developed that exploit the DOM[2] structure of the HTML pages. XPaths (discussed later, in the section on the Bag of XPaths model) have been used for pointing to and highlighting items of interest on Web pages. Myllymaki [9] describes ANDES, a software framework that uses XML technologies such as XHTML and XSLT to extract data from HTML pages. Webviews [10] allows users to create customized views of pages they wish to view, using XPaths. Reference [11] studies the robustness of item location in Web pages when different kinds of XPaths are used to point to these locations. Robustness under changes to the Web page is studied for different kinds of XPath expressions such as single-node pointing, alternative predicate expression, and relative addressing expression. Annotea [12] is a scheme for annotating documents on the Web and sharing such annotations, using XPointer, XLink, and HTTP. Xwrap [13] is an XML-enabled wrapper construction system for semi-automatic generation of wrapper programs. Here, the metadata about the information content that is implicit in the original Web pages is extracted and encoded explicitly as XML tags in the wrapped documents. The Lixto system [14] uses a new logic-based declarative language called Elog for extracting the specification pattern. The authors also provide a visual user interface (UI) for creating the wrapper programs. However, they use proprietary languages for wrapper generation and therefore cannot use the normal HTML browser for their visual interface. Most of these approaches assume that it is possible, by some means, to group similar pages into a cluster and to learn the structure of a prototypical page of each cluster. In the case of XML pages, knowledge of the data type definition (DTD) helps in identifying structurally similar pages. However, pages on the Web do not publish their DTDs. Although some induction algorithms exist to learn the DTDs [15] from a few examples, these assume that all of the pages on the site conform to the same DTD; the method fails when the pages are generated from different DTDs. Also, in most cases, the wrapper generation and the visual pointing and highlighting have been developed as standalone techniques that are to be tailored for different applications. ChangeDetector [16] is an end-to-end site-level monitoring tool that focuses on gathering "silent" information from corporate Web sites (e.g., changes in a competitor's organization structure and changes in a product line). Our system is also an end-to-end monitoring tool, which handles the information-extraction problem as follows:

1. Provide graphical user interfaces (GUIs) for the user to specify items of interest.
2. Identify structurally similar pages by comparing the XPaths of the pages.
3. Learn which elements to extract from each type by learning, from one sample, the XPaths for the various items of interest in that page.

This has the advantage that the solution is readily extendible to all sites for which the different pages are generated from a standard set of layouts or structures, as is the case on most commercial sites. Further, we have provided an end-to-end system which allows different users to request different items of information to be monitored and which generates reports for users on the basis of their individual requirements.

3. Architecture of the eShopmonitor

As mentioned in Section 1, the eShopmonitor is an independent standalone system that has been built to monitor the Web pages published on specified Web sites, to track changes and perform consistency checks to detect any errors, and to report on its findings. **Figure 1** shows the high-level block diagram of the system. The eShopmonitor supports users at two levels: the administrator and normal users. The administrator can configure the crawler and the miner and execute the whole system; a normal user can only specify queries and execute them.

Central to the eShopmonitor is the data store, which is the repository of the configuration and runtime information for the various components. Crawler configuration information, mined information, navigational information, configuration for the miner, subsequent crawled data, queries to be run, and reports generated by the queries—all reside in the data store, which comprises both flat files for the crawled data and a relational database for much of the other information.

The crawler, specially built for the eShopmonitor, can crawl both static (HTML files) and dynamic pages.

[2] The Document Object Model is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of documents. The document can be processed, and the results of the processing can be incorporated into the presented page. For further details, refer to *http://www.w3.org/DOM/*.

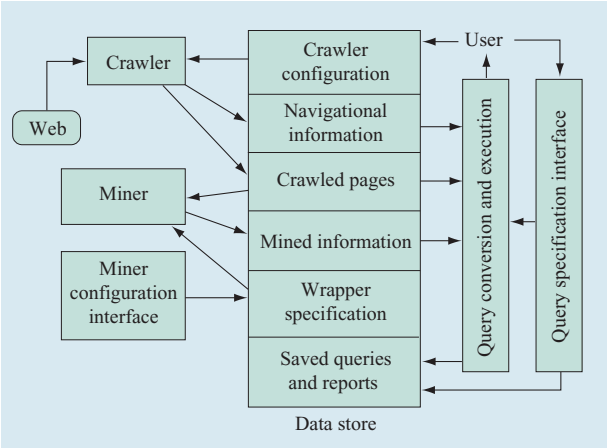

Figure 1

Block diagram of the eShopmonitor.

Dynamic pages are generated by some server code such as Active Server Pages (ASP), JavaServer Pages** (JSP**), or servlets at the time of invocation.

The various runtime parameters of the crawler may be configured by the administrator of the system, as explained in detail subsequently. The crawler crawls the Web pages being monitored at specified intervals, as mentioned in the crawler configuration. The crawled pages are dumped into the data store. The miner must initially be configured by the administrator for the different fields of interest to be mined in the different kinds of pages. This configuration information is again contained in the data store. These fields are subsequently mined from the crawled pages and are placed in the data store. The reporter component runs queries on the mined data on the basis of queries specified by the different users and system-level queries specified by the administrator. The queries are specified using the query interface and are stored in the system in the data store. In subsequent sections, we explore each of these components in detail.

4. Crawler

While many commercial crawlers were available, we required a crawler that could crawl both static and dynamic pages and that could also crawl password-protected sites where authentication was available. The crawler used was fully developed in-house and tuned to the specific requirements of commercial sites. On a functional basis, the requirements for the crawler are that the crawler should consistently be able to crawl an entire Web site and update the data store, and that mining should be completed and reports generated and transmitted to the users before the hours of peak activity on the site. The crawler must also be able to crawl dynamic pages which constitute a major part of any commercial site such as *ibm.com*. The crawler handles this by simulating form submissions by means of an automatic procedure for filling and submitting forms. It handles drop-down menus, radio buttons, and checkboxes; however, it cannot handle text fields and JavaScripts** when simulating form submission.

The parameters that define the functioning of the crawler include the following:

1. Seed URLs—the list of URLs from which the crawler begins its crawl.
2. Domains to crawl (and depth for each domain).
3. Location of the data store where the crawled files are stored.
4. File extensions to be included or excluded.
5. Password-protected seed URLs.
6. Crawl start time and reschedule frequency.

All of these values may be set by the eShopmonitor administrator through a Web-based interface provided for the crawler. The various configuration parameters are stored in an XML file which is read by the crawler at start time. Further, the user may also stop and start the crawler using the interface provided.

A crawl can be restricted to the pages of interest by configuring the crawler with particular domains, depths, and URL patterns to be included and excluded. Password-protected sites are also crawled by simulating form submission if the authentication information is available. Image crawling has not been included, since, according to our observations, an average *ibm.com* Web page contains six to eight images; crawling that many images synchronously with the Web pages degrades the overall running time and performance significantly. However, when needed, it may be included quite easily.

The crawler is multi-threaded and written in Java**. In order to avoid crawling restricted pages on a server, the eShopmonitor crawler adheres strictly to the robot exclusion protocol. Once a page is crawled, it is passed on to the miner for further mining. Meta-information about the page such as URL, HTTP status code, out-links, size of page, and depth are stored in the data store, as described later.

5. Miner

The main function of the miner is to mine the crawled pages, extracting items of interest from them. Assuming that the Web pages conform to the DOM standards, all dynamic Web pages generated by the same servlet (or any server-side code) have a similar DOM structure. Different groups of pages on the Web sites map to similar DOM structures, and such pages are said to belong to a template. A template could be an already prepared master

HTML page format that is used as a basis for composing new Web pages, or it could be server-side code which generates an HTML page in response to a user request, by dynamically connecting to some back-end database. The content of the pages may be dynamically generated, and the presentation is as specified by the template, resulting in a collection of pages that share a common look, feel, and structure.

In all pages generated from the same template, content such as the product name and the product price is present at locations which follow a pattern. An example is the search results presented by Google**. Whether a search returns two pages or ten, the locations of the title and summary of the returned Web pages follow a pattern.

In each template, a user may be interested in different items. We refer to each logically related set of items of interest as a bundle, and we allow several bundles to be associated with a template. For example, in a 'ProductDisplay' template, there can be a '(ProductName, ProductPrice)' bundle as well as a '(PromoDescription, DiscountPercent)' bundle. Note that, in a given Web page, there can be multiple instances of a given bundle.

For each template, the administrator must also define the bundles associated with the template and the fields of interest for each bundle, using the interfaces provided. Further, the administrator must also specify, for each template, a sample URL corresponding to that template. The interface also allows the addition of new templates and the deletion of existing templates. When the bundles have been defined for a template, on subsequent mining operations, the miner extracts the specified items of interest from all of the pages belonging to the template. This data is then passed to the store manager, which is the component that stores the data in the appropriate format. On *http://www.ibm.com/products/us* there are about 20 templates which generate 10–15K commercial pages of interest.

The miner has a multithreaded architecture. It has been designed to run independently and may also be invoked via a programmatic interface. It has a mechanism by which it can transfer the mined information to other programs or components (in our case, the data store). It processes *ibm.com* pages at an average rate of approximately 40 documents per second on a 2-GHz machine.

Bag of XPaths model

Previous systems such as XWrap and Lixto have defined their own language to write or generate template specifications. We have used the standardized XPaths as our wrapper language. We have found that XPath has sufficient expressive power to capture all types of generalization required to extract data reliably.

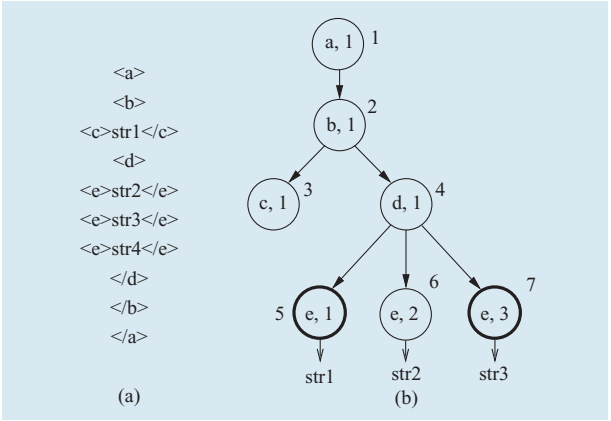

Figure 2

(a) An XML document and (b) its tree representation.

An XPath is a path expression that locates nodes in a DOM tree. We use only some of its features for our system; we explain our simplified version here. We consider each XPath as a sequence of terms separated by a slash (/). The syntax for each term is as follows:

$$nodetest[predicate]. \quad (1)$$

Here *nodetest* is a label defining a set of nodes (which we call a *node-set*) in which each node is a child node of the current node that has nodetest as its label. A predicate, which filters the node-set specified by the nodetest further into a smaller node-set, is always placed inside a pair of square brackets; [*position*( ) = *index*] and [*position*( ) < 7] are examples of predicates. We abbreviate predicates of the form [*position*( ) = *index*] as [*index*].

We call a predicate an *equality predicate* if it is of the form [*position*( ) = *index*]. We call other predicates *generalized predicates*. We call an XPath an *equality XPath* if all of the terms in the XPath contain only equality predicates. If some of the terms in an XPath contain generalized predicates, we call it a *generalized XPath*. **Figure 2** shows an XML document and the corresponding DOM tree. In this figure, the XPath for node 3 is /*a*[1]/*b*[1]/*c*[1]. For node 7, the XPath is /*a*[1]/*b*[1]/*d*[1]/*e*[3]. Both of these XPaths are equality XPaths. On the other hand, the XPath of the form /*a*[1]/*b*[1]/*d*[1]/*e*[(*position*( ) − 1) *mod* 2 = 0] is an example of a generalized XPath which evaluates to the node-set containing node 5 and node 7. Note that the last term of the XPath contains a generalized predicate.

Wrapper generation

A user is likely to be interested not only in properties or values of specific items in isolation but also in their

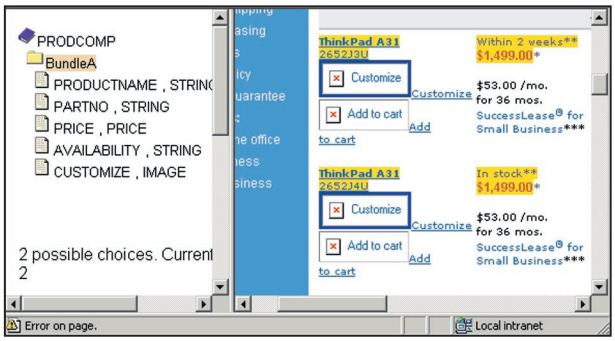

Figure 3

Screen shot for miner configuration. Reprinted with permission from [17]; © 2004 IEEE.

associations with other items. An example of such a query is "List all pages where 'ProductName' = 'ThinkPad A Series' and 'ProductPrice' < 1500." As stated earlier, a logically related set of items in a page is a bundle, and a page may contain more than one bundle—for example, a page containing prices and descriptions for ten products. Teaching the miner what to extract from each page is done in two steps:

1. The administrator specifies a sample page for the template to be mined.
2. The system presents the user with an interface consisting of two frames displayed side by side, with the sample page displayed on the right and an interface for defining the bundles, the elements of each bundle, and their names and types in the left-hand frame [see Figure 3 (shown later) for an example]. The association between the element name in the left frame and the item to be mined in the right frame is performed by requiring the user to select two consecutive examples of the item to be extracted. If there is only one instance of the item to be mined, the user clicks on the same instance twice; in this case, XPath is not generalized, but an absolute/equality XPath is generated. Internally, the miner then generalizes these locations as XPaths of these data locations, and thus learns the location pattern of items of the same type for that template. For example, if the user clicks on two items of the same type with XPaths /*html*[1]/*body*[1]/*table*[1]/*tr*[2]/*td*[1] and /*html*[1]/*body*[1]/*table*[1]/*tr*[5]/*td*[1], the XPath /*html*[1]/*body*[1]/*table*[1]/*tr*[(*position*( ) − 2) *mod* 3 = 0]/*td*[1] represents the generalization which, when evaluated on the pages of the same template as the example page, will return all items present in the page following this pattern. Other properties of the bundle elements, such as the name and type, are also specified through this interface. In practice, this generalization covers most of the items present in commercial sites, and we have observed high levels of accuracy in the data extracted.

One issue in template specification is the presence of optional items on a page. Some pieces of information might not always be present on all pages derived from the same template. For example, multiple pages listing personal computers might be generated from the same template. However, only some of the pages might have a markup indicating a special sale on some models. In such cases of optional items, another kind of XPath generalization, based on the count of sibling and child nodes, has been used. For example, consider the absolute XPath /*html*[1]/*body*[1]/*b*[1]/*a*[1] that corresponds to a field of interest. Assume that an optional 'For Sale' tag has been added as an anchor tag (<*a*>) before this field. In that case, the generalized XPath used is /*html*[1]/*body*[1]/*b*[1]/ *a*[(*count*(../*a*) > 1) *and* (*position*( ) = 2) *or* (*count*(../*a* = 1) *and* (*position*( ) = 1)]. This XPath evaluates to the first anchor if the optional tag is absent and the second XPath if the optional tag is present.

Note that through these two generalized XPaths [viz. *position*( )−*x mod y* = 0 and *count*(..)], we can capture nested for-loops and if-then-else statements in the server-side code that generated these pages. Together, these constructs are sufficiently rich to express a wide variety of templates, a fact which is confirmed by our experiments on various e-commerce sites (See Section 7).

As a specification example, consider **Figure 3**, which shows the user interface. The highlighted fields are ones that have been added to the configuration for extraction after every mining operation.

The interface also handles the cases in which more than one generalized XPath is possible. In that situation, the user is presented with a choice list of those generalized XPaths and can choose the most correct one.

This template specification exercise has to be done only once while installing the system, and the entire exercise for six different templates on *ibm.com* takes about 15 to 20 minutes. The administrator has to reconfigure the system only when the templates undergo a drastic change. The eShopmonitor has the capability to alert the administrator to such a scenario. Small changes to the template are automatically handled by the eShopmonitor and require no human intervention. Details are presented in the section on handling changes to the template.

Data extraction

We perform various tasks before extracting the required information:

1. Clean the HTML pages: We have noted that not all HTML pages are well formed, like XML. Further, the HTML is not always well written; many pages have missing closing tags, incorrect nesting of tags, and other issues. While many browsers are sufficiently rugged to display the pages properly, we need to clean up the HTML pages before identifying the XPaths for the elements, since our processing assumes that the document is an XML document. We have used JTidy[3] to clean the HTML pages, balance tags, and tag the data in XML. A DOM is obtained after this step.
2. Identify the page/template type: Crawled pages are either static or dynamic. For a dynamic page, the template which generated the page is detected from the URL–template mapping (see the discussion below.)
3. Extract the items of interest: For each template, the data store contains the set of XPaths to be mined, which map to the user's fields of interest. Hence, for each dynamic page, the miner extracts the fields of interest.
4. Process different data types with the Type Handler: Evaluation of a given XPath on the documents returns a DOM node. It is then processed by the appropriate type handler to extract fields from the items (discussed later). The Type Handler system currently supports types such as numeric, string, long string, price (contains attributes value and currency), image, and meta.

The data-extraction procedure is outlined in the mining algorithm shown in **Figure 4**.

```
Input : Set of crawled pages P, templates T
Output : Mined information in the database
Algorithm :
∀T ∈ T {
    Errors[T] = 0;
}
∀d ∈ P {
    d = CleanHTML(d);
    T = Template(d);
    if (T = NULL) skip to next document;
    /* T = NULL means it is a static page */
    C = TemplateConfig(T);
    bundle_id = 0;
    ∀bundles b ∈ C{
        B = ExtractBundleInstances(b, d);
        if (B = NULL) Errors[T] + +;
        ∀instances b_i ∈ B{
            ∀elements e ∈ b{
                node = ElementInstance (b_i, e);
                if (node = NULL) Errors[T] + +;
                else {
                    x = TypeHandle(node; e);
                    if (x = NULL) Errors[T] + +;
                    else Save(x, e, d, bundle_id);
                }
            }
            bundle_id + +;
        }
    }
}
∀T ∈ T {
    if (Errors [T] > threshold)
        RaiseException("Template Changed");
}
```

Figure 4

Data extraction procedure.

Template–URL mapping/template clustering

To be able to extract the items of interest from the pages, it is necessary to know the template from which the page is derived. For *ibm.com*, the information contained in the URL of the page, along with the necessary database tables which contain the template–URL mapping, allows us to identify a page template by reading its URL. However, the template–URL mapping information might not be generally available. Therefore, a classifier is required to map pages to templates on the basis of a similarity measure between semi-structured documents. We have developed an algorithm (discussed in a separate work [18]) to measure the structural similarity of semi-structured documents. The algorithm works very accurately on a variety of Web sites (IBM, Dell, Amazon, etc.). Henceforth, we can assume that the URL-to-template mapping is given to us.

Handling changes to the template

One issue that is of great concern for wrappers is their maintainability. The structure of a Web page may change at times, and the wrapper for it may require certain modifications for its desired extraction process. In this subsection, we outline a method by which we can construct new XPaths on the new template (based on the old XPaths of the old template) that will extract the desired fields of interest in the modified template. Here we assume that the structural changes to the template are only slight. If drastic changes to the template occur, the mining algorithm will raise an exception caused by too many misses or type-handling errors (see Figure 4).

Roughly, the approach is as follows. We search for the fields of interest (whose values are known in the old template) in the new template by means of wild-card XPaths. By this means we obtain the locations of the fields in the new template and their corresponding XPaths. Since the XPaths are assumed to have changed only slightly for a field, we assign an XPath "closest" to its old XPath. Formally speaking, let T_{old} and T_{new} be the DOM trees of a page corresponding respectively to its

[3] *http://sourceforge.net/projects/jtidy/*.

Table 1 PRODUCTPRICE table.

Name	*Type*
URLHASH	CHAR(32)
BUNDLEID	INT
PRICEVALUE	DOUBLE
CURRENCY	CHAR(3)

old and new templates. Let p^i_{old} denote an equality XPath in T_{old} for a node with value v^i_{old}. We construct a new XPath $p_{\text{new}} = //*/v^i_{\text{old}}$ and evaluate on T_{new}. Let the set $X = \{x^i_{\text{new}}, \cdots, x^k_{\text{new}}\}$ denote the k nodes that are returned when p_{new} is evaluated on T_{new}. We create an equality XPath p^i_{new} for each x^i_{new} where $1 \le i \le k$. We then choose the p^j_{new} as a new XPath for the node corresponding to the value v^i_{old} that has the least edit distance with the old XPath p^i_{old}, i.e.,

$$NewXPath(v^i_{\text{old}}) = \text{argmin}_{p^j_{\text{new}}} EditDist(p^j_{\text{new}}, p^i_{\text{old}}). \quad (2)$$

The intuition for doing so is that since there are typically only slight changes in the structure of the templates, the new XPaths corresponding to the old XPaths are very similar and therefore have the minimum edit distance.

Type handling

An item may require more than one field for complete specification. For instance, price is specified by an amount and a currency. The type-handling package within the miner appropriately parses the items and extracts the attributes from the items. The eShopmonitor supports two kinds of types, simple and composite. Simple types are int, double, string, and long string. When an item is specified by more than one attribute, it is called a composite item. An example is the image tag which has "src" and "alt" as two attributes. These items are extracted as one string, and it is the responsibility of the appropriate type handler to parse and extract various attributes from the items.

Currently the eShopmonitor has a number of type-handlers such as PRICE, LEASEPRICE, IMAGETAGS, METATAGS, and a GENERIC type which has two fields, a numeric and its unit. Generic type covers all data items such as 1 GHz and 256 MB. A new type can be easily introduced into the system as a plug-in.

For handling some composite types such as PRICE, additional information is required. For example, PRICE contains a CURRENCY attribute whose value can be USD (U.S. dollars) CAD (Canadian dollars), etc. To obtain the actual value, the PRICE type-handler uses hints from the URL (whether it is from a domain such as *www.ibm.com/us/*), hints from the language (a French-Canadian page is likely to show Canadian dollars), and the writing style of the prices (e.g., 1500,00$ is a Canadian price and $1,500.00 is an American price).

Data store

Database schema

At the top level of the store design are unique identifiers for each crawl (referred to subsequently as the *crawlid*) and for each URL crawled (which is stored as a hash of the URL and henceforth referred to as *urlhash*). Further, each bundle instance is also uniquely identified by a bundle ID, referred to as *bundleid*.

As described in greater detail in Section 6, a query may specify one or more conditions. A condition is a single constraint such as "where PRODUCTNAME contains 'ThinkPad'." These base conditions may be joined by the operators ANDWHERE, AND, and OR. The operator ANDWHERE between two conditions specifies that the items should be selected from the same bundle. In contrast, the operator AND specifies that the items could be selected from anywhere in the same page. Further, it is possible to add or delete a new item during the course of use. To support this feature, each item has its own table, listing the URL, the bundle ID, and the value of the item (the last of which is determined also by the type of the item.) For example, for the item PRODUCTPRICE of type PRICE, we have a table called PRODUCTPRICE appended by the crawlid, and columns as shown in **Table 1**.

Store manager

After extraction and processing based on the type, the items are stored in a relational database. As earlier, the miner passes the mined data to the store manager. The store manager then processes the bundles and translates them into the appropriate SQL statements for insertion into the relevant tables. It caches the data and then dumps it into the database when the cache is full. Since this component is thread-safe, the miner threads call it in asynchronous mode.

6. Reporter

When the crawling and mining are completed, the reporter executes queries on freshly extracted data. All user-specified queries are executed, and a single consolidated report is then sent to the user.

Reporter subsystems

The reporter component comprises the following subsystems:

1. A query-specification interface for specifying various kinds of queries: Currently the eShopmonitor has support for four different kinds of queries, which are

explained later in this section. Using the query-specification interface, a user can save his query or run it online. Saved queries are executed later in batch mode, and a consolidated report is sent to the user.

2. Query conversion modules: Since the crawled and mined information is stored in DB2* tables, any query on the system must be translated into SQL. The query-conversion modules convert the queries, as specified by the user through the interfaces, into SQL.

   Note that instead of defining our own query-conversion modules, on-line analytical processing (OLAP) and data warehousing tools could have been used. However, one major concern during system design was to make the query-specification interfaces very easy to use and to the point. This necessitated the design of an intermediate language and query-conversion modules that converted queries from this intermediate language to SQL. Note, however, that multidimensional queries (as in OLAP) are fully supported by this intermediate language (see the section on change and content query formulation).
3. Query-execution modules: Since the different query types are quite varied in nature, the eShopmonitor has a set of execution modules, one for each query type. The query execution controls the execution of the query and also determines the display of the query results.
4. Query verifier: Since the primary aim of the eShopmonitor is to detect discrepancies, it is natural to verify any anomaly reported by the eShopmonitor. For this purpose, every query result is linked to its corresponding query verifier module. The relevant result can be verified by using that module. For example, for a price change query, the verifier shows the old and new versions of the page and highlights the price change.

Query types

Various types of queries are supported in the eShopmonitor. These queries use the element names of the various bundles that have been specified while configuring the miner. In addition, navigational information and some parameters for each URL such as its signature and status code are also stored. These parameters can be queried, and special queries such as those on meta tags are also supported. In general, all queries return a list of URLs that match the criteria specified in the query. Broadly speaking, a query is of one of the following types:

- *Content queries:* As the name suggests, these queries allow the user to pose queries on the content of the pages, e.g., "List all pages where the PRICE is less than $500" and "List all pages where the PRODUCT_DESCRIPTION contains "Now much cheaper." Such queries refer to only one crawled version.
- *Change queries:* These queries look for changes in page content across two different crawls, e.g., "List all pages where PRICE changed by $400 or more." These queries compare the contents of a page across two crawled versions. To arrive at meaningful results, it is ensured that some defining entity such as PARTNO is also compared while inter-version comparisons are being performed. For example, there might be two prices on a page, $400 and $410, corresponding respectively to part numbers A and B. While comparing another version of the same page, we avoid comparing the prices of A and B (using part numbers). Currently, the defining entity is set to PARTNO, and all change queries report the PARTNO along with the results.
- *Navigational path queries:* These queries are used to compute the best path between two given URLs. Here, the "best" path is the one that has the fewest intermediate links. If several paths are best, the first path is returned. For computational reasons, the eShopmonitor restricts the maximum path length to 5. If no such path is found, the URLs are listed as disconnected.

  These kinds of queries can be used to verify whether or not two sub-sites are connected via a short path of anchor links. For example, if the ThinkPad* and Optical Drive sub-sites are related, the administrator would prefer short paths between them. For this purpose, navigational path queries can be used.
- *Navigational link queries:* These queries are parameterized with three attributes—a URL (U), an in-depth (I), and an out-depth (O). The query shows all paths that have a length I and end at U, and paths that begin at U and have a length O. The former class of paths is called the *in-tree* of U and the latter is called the *out-tree*, hence these queries are also called navigational tree queries. For computational reasons, there is an upper limit on I and O.
- *Fixed queries:* These queries do not fall into any specific category and do not follow any particular rules. There is no converter to translate these queries. However, an administrator with the knowledge of the internal design (and hence the SQL) can add fixed queries through an interface provided by the eShopmonitor. Many fixed queries exist as default queries in the system, e.g., "List all missing URLs" or "List all HTTP 404 URLs."

Notes: 1) For ease of use, change and content queries are allowed to be mixed. 2) META tag queries are classified as content queries.

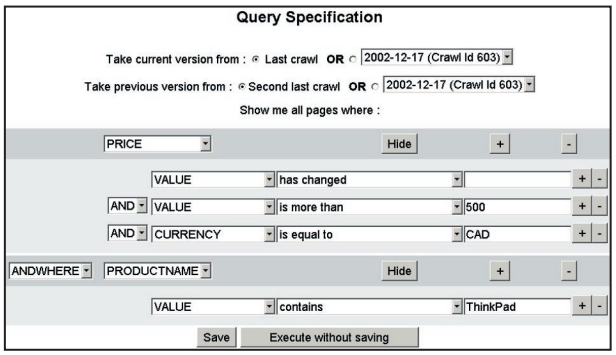

Figure 5

Specification of the query "Show me all pages where the price has changed and where the price is more than CAD 500 and where product name contains ThinkPad."

Change and content query formulation

Each change and content query is made up of conditions that are joined by the connectors AND, OR, and ANDWHERE. Each condition refers to exactly one mined attribute. The general syntax of a condition is *Element.attribute operator value*. For example, PRICE is a composite element and has two attributes—CURRENCY and VALUE. Therefore, we have conditions such as "PRICE.CURRENCY = USD." Since VALUE is a default attribute present in almost all elements (except IMG and META), we sometimes use expressions such as "PRICE < 100" instead of "PRICE.VALUE < 100." If C1 and C2 are two such conditions, the semantics of C1 [connector] C2 is given by the following:

- Connector = AND: A page where C1 and C2 hold (anywhere on the page) is a valid result.
- Connector = OR: A valid result page should have elements that satisfy either C1 or C2 or both.
- Connector = ANDWHERE: C1 and C2 should hold inside the same bundle (refer to Section 5 for the concept of a bundle). For example, if C1 = 'PRODUCTNAME = ThinkPad' and C2 = 'PRICE.VALUE < $1000', C1 ANDWHERE C2 will return pages that display less expensive ThinkPads. Had we used C1 AND C2, we would have obtained erroneous pages, such as pages that display ThinkPads and also $10 cables.

All conditions use exactly one operator such as *is less than* and *contains*. One special operator, *is anything*, is a wild card which matches everything. It can be used to display additional attributes in the query results without adding any extra conditions. **Figure 5** illustrates specification of the query "Show me all pages where (PRICE.VALUE > 500 AND PRICE.VALUE has changed AND PRICE.CURRENCY = 'CAD') ANDWHERE (PRODUCTNAME contains 'ThinkPad')."

Query conversion

Change and content queries are converted into SQL as follows. Let us consider a query of the form C1 [connector] C2, where C1 and C2 are clauses. Each clause refers to exactly one field of interest, which is represented by a table in the data store; hence, in the final SQL, it contributes a clause. For example, PRICE.VALUE < $400 corresponds to "Select * from price_<crawl version> where value < 400."

If there are multiple conditions, one SQL clause is made for each of them. If the connector is AND or OR, the clauses are connected using the SQL keywords INTERSECT and UNION, respectively.

For example, "PRICE.VALUE < $400" AND "NAME = Printer" corresponds to "(select * from price_<crawl version> where value < 400) intersect (select * from name_<crawl version> where value = 'Printer')."

If the connector is ANDWHERE, instead of using intersects or unions, we do an SQL inner join of the respective tables. The join condition is that the bundle IDs should match.

For example, "PRICE.VALUE < $400" ANDWHERE "NAME = Printer" corresponds to "select * from price_(crawl version>, name_<crawl version> where value < 400 and value = 'Printer' and price.bundleid = name.bundleid." This scheme can easily be extended to multiple conditions.

Navigational queries are converted first into a breadth-first search routine on the crawl graph; then the parameters of that routine are plugged into a corresponding SQL.

Query execution and verification

The query is executed by running its SQL against the database. The query results obtained are displayed in the form of a table. Typically, each result contains a list field. It means that all of the displayed URLs satisfy the specified conditions.

For verification purposes, each URL, when clicked, leads to a verifier. It is meant to cross-check the query results. The most important verifier is that for change queries. Change query verifiers display the old and the changed versions of the clicked URL. The text deleted in the old page is highlighted in red and the text inserted in the new page is highlighted in yellow so that any changes can be quickly identified.

Figure 6 shows a sample output for a change query; in this case, the query is "Show me all pages where the price has changed and where the price is more than $500." The

URLs in the second column are the pages where the price has changed and the price is more than $500. By clicking on any of these URL links, the actual page contents, both old and new, are displayed, as shown in **Figure 7**, which is the verifier for the change query. This figure is a screen shot of the results of the change query, with the latest version being displayed in the top frame and the earlier version being displayed in the bottom frame.

Report generation

Each user can save the specified queries. These queries are executed at the end of the crawl-mine-report cycle of the eShopmonitor. Query results from each of the saved queries are collected and saved in an HTML report file. For performance purposes, each query that can potentially be shared among many users is executed only once, and its results are cached for subsequent users. A summary of all of the queries is generated and mailed to the user, along with a link to the detailed consolidated report file.

Thus, the user receives a daily digest of the reports in which he is interested, and he can easily view and verify the individual results.

Report configurator

This module provides the user with an interface by which he can add or delete reports that may be scheduled. The eShopmonitor supports two types of reports:

1. Fixed reports: The underlying queries for such reports are predefined. Examples of fixed reports include
   - Lists of URLs whose contents have changed since the previous crawl.
   - Lists of URLs in which any field of interest (i.e., product price, product, and description) information has changed.
   - Lists of broken links.
2. Parameter-based reports: The queries for these reports require a set of parameters, which is provided by the user. Parameter-based reports are of three different types:
   - Navigational reports: Examples are
     (a) All pages which point to the given page x.
     (b) All pages which are pointed to by the given page x.
     (c) All pages which are reachable in y steps from the given page x.

     In the above three examples, the values of x and y are provided by the user through the report configurator user interface.
   - Content-based reports: Examples are
     (a) All pages where price is less than $\$x$.
     (b) All pages where product name is X and price is less than $\$y$.
   - Change-based reports: Examples are
     (a) All pages where price has changed by more than $\$x$ since yesterday.
     (b) All pages where product description for X has changed since last week.

eShopMonitor : Report Page

Query : Show me all pages where (PRICE has changed) ANDWHERE (PRICE is more than 500)

Current version from : 865 Previous version from : 864

S.No.	URL	PARTNO	NEW_PRICE	OLD_PRICE
1	United States | ThinkPad T30	236641U	1749.0	2099.0
2	United States | IBM Infoprint® 1140 Laser Printer	4540DN1QC	2073.0	2270.0

Summary : 2 results, 2 distinct URLs.

Figure 6

Listing of results for a change query, "Show me all pages where the price has changed and where the price is more than $500." Reprinted with permission from [17]; © 2004 IEEE.

Figure 7

Screen shot of the display of the two URLs for a change query. Reprinted with permission from [17]; © 2004 IEEE.

Online query interface

This module provides a mechanism for generating on-demand reports. Note that the reports are generated using the last crawled data, and no data fetching is needed to

Table 2 Mining statistics across various e-commerce sites.

Dataset	*Pages*	*Templates*	*Bundles*	*Non-empty fields*	*Empty fields*	*NULL fields (%)*
IBM	3,532	6	13,102	51,693	3,415	6
Dell	3,747	6	12,996	41,214	7,564	15
Amazon	205	2	651	2,044	268	12

generate these reports. This interface is meant primarily for users to test out newly added queries immediately.

In addition to the components described above, the eShopmonitor also has a user component with which the administrator may define the various users of the system, assign passwords, and define level of access, in terms of the user type. Individual users may also view user information and modify user information by means of the user interface provided.

The entire system has been built on a Linux** platform, and all of the functionality is viewable using IE 5.5 or above. The data store is built on DB2 UDB 7.2. This system has been completed and pilot-tested at the *ibm.com* site.

7. Evaluation

We evaluated the eShopmonitor on three sets of pages crawled from the three large online commercial shops. First we clustered the documents on the basis of structural similarity using the algorithm in [18]. We manually selected a set of clusters and specified the wrappers using the miner configuration interface. Details of the data set and results are given in **Table 2**.

A field of interest in a page is considered to be NULL if either tag is not present or if it does not match the expected type. For example, if a string occurs instead of a PRICE, we save it as NULL. Table 2 gives the number of NULL and non-NULL fields found on the pages.

A field of type PRICE is present in every template in IBM commercial pages. The eShopmonitor is able to parse a high percentage (96.7%) of the string evaluated by the XPaths for PRICE (12,659 of 13,102). This points to the fact that most of the bundles identified were indeed correct.

8. Conclusion and future work

In this paper, we have presented the architecture and design of the eShopmonitor, an independent tool for monitoring data of interest at commercial sites. It is an end-to-end solution comprising a highly configurable crawler, a miner that learns items of interest with just one example for each type of page, and a very comprehensive reporting system. It uses the underlying DOM structure of the tidied HTML pages, both to identify types of pages and extract fields of interest, with a novel use of XPaths.

Another novel feature is the user interface for configuring the miner, which allows the user to select items of interest by simply specifying a sample page and clicking on the data items of interest on that page. Though it was initially designed for *ibm.com*, extensions have been added to enable it to monitor other sites as well. Currently work is in progress to automate discovery of bundles in a page, or discover interesting items to mine in a page, which may then be presented to the user for confirmation.

References

1. N. Kushmerick and B. Thomas, "Adaptive Information Extraction: Core Technologies for Information Agents," in *Intelligent Information Agents R&D in Europe: An AgentLink Perspective*, Springer-Verlag, New York, 2002.
2. N. Kushmerick, "Wrapper Induction: Efficiency and Expressiveness," *Artificial Intelligence* **118,** No. 1/2, 15–68 (2000).
3. C. A. Knoblock, K. Lerman, S. Minton, and I. Muslea, "Accurately and Reliably Extracting Data from the Web: A Machine Learning Approach," *IEEE Data Eng. Bull.* **23,** No. 4, 33–41 (2000).
4. G. Huck, P. Fankhauser, K. Aberer, and E. J. Neuhold, "Jedi: Extracting and Synthesizing Information from the Web," *Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS),* 1998, pp. 32–43.
5. V. Crescenzi, G. Mecca, and P. Merialdo, "Roadrunner: Towards Automatic Data Extraction from Large Web Sites," *Proceedings of the 27th Very Large Database (VLDB) Conference,* Rome, Italy, 2001, pp. 109–118.
6. J. Hammer, H. Garcia-Molina, J. Cho, A. Crespo, and R. Aranha, "Extracting Semistructured Information from the Web," *Proceedings of the Workshop on Management of Semistructured Data,* 1997, pp. 18–25.
7. B. Adelberg, "NoDoSE—A Tool for Semiautomatically Extracting Structured and Semistructured Data from Text Documents," *Proceedings of the ACM SIGMOD Conference on Management of Data,* 1998, pp. 283–294.
8. C.-H. Chang and S.-C. Lui, "IEPAD: Information Extraction Based on Pattern Discovery," *Proceedings of the 10th International Conference on the World Wide Web,* ACM, 1-58113-348-0/01/0005, 2001.
9. J. Myllymaki, "Effective Web Data Extraction with Standard XML Technologies," *Proceedings of the 10th International Conference on the World Wide Web,* ACM, 1-58113-348-0/01/0005, 2001.
10. J. Freire, B. Kumar, and D. Lieuwen, "Webviews: Accessing Personalized Web Content and Services,"

Proceedings of the 10th International Conference on the World Wide Web, ACM, 1-58113-348-0/01/0005, 2001.
11. M. Abe and M. Hori, "Robust Pointing by XPath Language: Authoring Support and Empirical Evaluation," *Proceedings of the IEEE Symposium on Applications and the Internet,* 2003, pp. 156–165.
12. J. Kahan, M.-R. Koivunen, E. Prud'Hommeaux, and R. R. Swick, "Annotea: An Open RDF Infrastructure for Shared Web Annotations," *Proceedings of the 10th International Conference on the World Wide Web,* ACM, 1-58113-348-0/01/0005, 2001.
13. L. Liu, C. Pu, and W. Han, "Xwrap: An XML-Enabled Wrapper Construction System for Web Information Sources," *Proceedings of the 16th International Conference on Data Engineering (ICDE),* 2000, pp. 611–621.
14. R. Baumgartner, S. Flesca, and G. Gottlob, "Visual Web Information Extraction with Lixto," *Proceedings of the 27th Very Large Database (VLDB) Conference,* Rome, Italy, 2001, pp. 119–128.
15. M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim, "XTRACT: A System for Extracting Document Type Descriptors from XML Documents," *Proceedings of the ACM SIGMOD Conference,* 2000, pp. 165–176.
16. V. Boyapati, K. Chevrier, A. Finkel, N. Glance, T. Pierce, R. Stokton, and C. Whitmer, "Changedetector™: A Site-Level Monitoring Tool for the WWW," *Proceedings of the 11th International Conference on the World Wide Web,* ACM, 2002, pp. 570–579.
17. N. Agrawal, R. Ananthanarayanan, R. Gupta, S. Joshi, R. Krishnapuram, and S. Negi, "eShopmonitor: A Web Content Monitoring Tool," *Proceedings of the 20th International Conference on Data Engineering (ICDE),* 2004, in press.
18. S. Joshi, N. Agrawal, R. Krishnapuram, and S. Negi, "A Bag of Paths Model for Measuring Structural Similarity in Web Documents," *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD),* 2003, pp. 577–582.

Received September 15, 2003; accepted for publication December 30, 2003; Internet publication August 31, 2004

Neeraj Agrawal *IBM Research Division, IBM India Research Laboratory, Block I, Indian Institute of Technology (IIT), Hauz Khas, New Delhi 110016 (nagrawal@in.ibm.com).* Mr. Agrawal received his bachelor's degree in computer science and engineering in 2002, joining the IBM India Research Laboratory that same year. He is currently pursuing a master's degree in computer science and engineering at the Indian Institute of Technology, Delhi. His interests are in information extraction, text mining, content monitoring, and Web mining.

Rema Ananthanarayanan *IBM Research Division, IBM India Research Laboratory, Block I, Indian Institute of Technology (IIT), Hauz Khas, New Delhi 110016 (arema@in.ibm.com).* Ms. Ananthanarayanan joined the IBM India Research Laboratory as a Research Staff Member in 1998 and became a member of the Knowledge Management group in 2002. She is currently working in the areas of automated information extraction and Web mining. Before assuming her current responsibilities, she worked in the areas of auctions, negotiations, and machine learning. Ms. Ananthanarayanan received a master's degree in computer science and engineering from the Indian Institute of Technology, Chennai, in 1994.

Rahul Gupta *IBM Research Division, IBM India Research Laboratory, Block I, Indian Institute of Technology (IIT), Hauz Khas, New Delhi 110016 (rahulgupta@in.ibm.com).* Mr. Gupta received a bachelor's degree in computer science and engineering in 2001, joining the IBM India Research Laboratory that same year. He is currently a member of the Knowledge Management group and is pursuing a Ph.D. degree in computer science and engineering at the Indian Institute of Technology, Delhi. His prior work involved media mining, similarity search, and relevance feedback. Mr. Gupta's current interests include Web mining, information extraction, site analysis, Web modeling, random graphs, and machine learning.

Sachindra Joshi *IBM Research Division, IBM India Research Laboratory, Block I, Indian Institute of Technology (IIT), Hauz Khas, New Delhi 110016 (jsachind@in.ibm.com).* Mr. Joshi is a Research Staff Member in the Knowledge Management group. He joined the IBM India Research Laboratory in 2000 after receiving a master's degree in computer science and engineering from the Indian Institute of Technology, Bombay. Since 2000 he has been working in the areas of Web mining, machine learning, and text mining. Mr. Joshi's recent work involves automatic data extraction from semi-structured documents and Web site categorization.

Raghu Krishnapuram *IBM Research Division, IBM India Research Laboratory, Block I, Indian Institute of Technology (IIT), Hauz Khas, New Delhi 110016 (kraghura@in.ibm.com).* Dr. Krishnapuram received his Ph.D. degree in electrical and computer engineering from Carnegie Mellon University in 1987. From 1987 to 1997, he was on the faculty of the Department of Computer Engineering and Computer Science at the University of Missouri, Columbia. In 1997, Dr. Krishnapuram joined the Department of Mathematical and Computer Sciences at the Colorado School of Mines (CSM), Golden, Colorado, as a Full Professor. Since 2000 he has been

with the IBM India Research Laboratory, where he currently manages the Knowledge Management group. Dr. Krishnapuram's past research encompasses many aspects of fuzzy set theory, neural networks, pattern recognition, computer vision, and image processing. He has published more than 150 papers in journals and conferences in these areas. He is an associate editor of the *IEEE Transactions on Fuzzy Systems* and a coauthor (with J. Bezdek, J. Keller, and N. Pal) of the book *Fuzzy Models and Algorithms for Pattern Recognition and Image Processing*.

Sumit Negi *IBM Global Services, IBM India Research Laboratory, Block I, Indian Institute of Technology (IIT), Hauz Khas, New Delhi 110016 (sumitneg@in.ibm.com).* Mr. Negi received a bachelor's degree in electronics and communication engineering in 2001. He joined IBM that same year and became a member of the Knowledge Management group at IRL in 2002. His interests lie in Web mining, focused crawling, and information extraction.