Comparative Mining of B2C Web Sites by Discovering Web Database Schemas

C.I. Ezeife
School of Computer Science
University of Windsor
Windsor, Ontario N9B 3P4
cezeife@uwindsor.ca

Bindu Peravali
School of Computer Science
University of Windsor
Windsor, Ontario N9B 3P4
peraval@uwindsor.ca

ABSTRACT

Discovering potentially useful and previously unknown information or knowledge from heterogeneous web contents such as "list all laptop prices from Walmart and Staples between 2013 and 2015 including make, type, screen size, CPU power, year of make", would require the difficult task of finding the schema of web documents from different web pages, performing web content data integration, building their virtual or physical data warehouse integration before web content extraction and mining from the database. Wrappers that extract target information from web pages can be manual, semi-supervised or automatic systems. Automatic systems such as the WebOMiner system, use some data extraction techniques based on parsing the web page html source code into a document object model (DOM) tree, then traverse the DOM for pattern discovery. Some limitations of these existing systems include using complicated matching techniques such as tree matching, Finite state automata, not yielding accurate results for complex queries such as historical and derived.

This paper proposes building the WebOMiner_S which uses web structure and content mining approaches on the DOM-tree html code to simplify and make more easily extendable, the web data extraction process of the WebOMiner system. The WebOMiner system is based on non-deterministic finite state automata (NFA) to recognize and extract web different types (e.g., text, image, links, and lists). The proposed WebOMiner_S replaces the use of NFA of the WebOMiner with a frequent structure finder algorithm which uses regular expression matching in Java xpath parser and methods (such as compile(),evaluate()) to dynamically discover the most frequent structure (which is the most frequently repeated blocks in the html code represented as tags $< divclass = \text{“ ''} >$) in the Dom tree. This approach eliminates the need for any supervised training or updating the wrapper for each new B2C web page making the approach simpler, more easily extendable and automated.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IDEAS '16, July 11-13, 2016, Montreal, QC, Canada
Copyright 2016 ACM 978-1-4503-4118-9/16/07 ...$15.00.
http://dx.doi.org/10.1145/2938503.2938522.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: [Database Applications, Data Mining]; K.6.5 [**Management of Computing and Information Systems**]: Data Extraction and Web Mining—*E-commerce software, automatic extraction*

General Terms

Web Content Mining, Automatic Web Data Extraction, Wrappers

Keywords

Web Content Mining, Automatic Web Data Extraction, Data integration, Wrappers

1. INTRODUCTION

The World Wide Web (Web) is a popular and interactive medium to disseminate pages into structured data, a lot of efforts have been devoted in the area of information extraction (IE) [2]. This involves using DOM trees ([2], [1]), Document Object Model trees which are the representation of underlying source html code of the web page and can be addressed and manipulated using methods on the objects. A web page of an e-commerce B2C website consists of several information (navigation, advertisements, copyright notices, shipping information, contact information) along with the products information. Every B2C website showcases its products arranged in a block by fetching them from underlying databases. In a product rich web page, all the products in it will be styled. According to W3 consortium [4], html tags such as $< div >$, $< table >$, $< tr >$, $< td >$, $< ul >$, $< ol >$, $< li >$, are block level tags and this list has up to 32 tags, but the above mentioned tags are those used to section a web page or group elements in a page. In a product list web page, all the product elements are grouped, each attribute of the product will be applied the same style all over the web page, e.g., all product images will be of same height/width, all product names will be in same font/color etc. This is done using the class attribute in the div tag expressed as $< divclass >$. Whenever content (such as product laptop tuples) in a webpage repeats, its underlying html source code also repeats, which means the same set of html tags will be repeated in the html code of web page with just the difference in the content instances between them. However, retrieving products information from webpages into structured data is a challenging task.

To mine data on web documents, the html files are usually converted to pre-parsed documents in a DOM (document ob-

ject model) tree by many systems ([2], [1]). In a DOM tree, tag and tag attributes are represented with nodes that specify the hierarchical relationships between tags. Formally, the comparative shopping agent given an input product list page P from any e-commerce web site needs to be able to first extract:
(i) its main table product schema T of page P, each with its n attributes a_{1p}, a_{2p}, ...i, a_{np},
(2) its m tuples of Table type T and Database type P, and
(3) store it in an integrated timestamped, historical relational database forming a data warehouse of products from several e-commerce web sites to accommodate comparative shopping and historical querying on web data.

1.1 Contributions and Outline

This paper proposes a web data extraction system called the WebOMiner_S (standing for the WebOMiner_Simple), which can discover the schema of any given set of Business to Customer (B2C) product web pages such as Walmart laptop web page. The WebOMiner_S follows the architecture of the WebOminer web data extractor system described in [6]. The architecture of the WebOMiner system is summarized below with its code downloadable from its journal web site at *http : //users.monash.edu.au/ dtaniar/IJDWM/*. The proposed system, the WebOMiner_S contributes to the problem of web data extraction and mining with the following enhanced major extensions to the WebOMiner system. The WebOMiner system goes through the four main modules of (A) Crawler module, (B) Cleaner module, (C) Extractor module and (D) Miner module in automatically extracting the table schemas and their tuples from different B2C web sites. In the first step, it takes local html source code of the web page, passes that to HTMLcleaner2-2 in the second step to clean and obtain well formed html file which is converted to DOM tree html web page in step 3. Then, Java algorithm is run to parse the DOM tree to identify the data body zone of the web page containing the relevant data to be extracted. This body zone is passed through a Java algorithm content extractor which extracts heterogeneous web content data that can be of any type such as text, image, list, form. These web contents are passed in step 4 through the non-deterministic finite state automatas (NFA) which will identify the specific content type and classify them into tuples of their type so that they are stored in the database. The NFA with the webominer system helps to increase the process of automation of web data extraction by automatically recognizing structures similar to those from the previously seen B2C web page structures. However, there is need to simplify the complexity of the method used for data extraction, need to make the system more available, more extendable and adaptable. These are the goals of the proposed WebOMiner_S.
1) We propose to replace the use of the NFA in step 4 of the WebOMiner system with a frequent structure finder (FSfinder) algorithm using both web content and structure mining. The FSfinder uses the Java xpath parser to summarize the frequency of each tag occurrence in the web html code, retaining only tags that meet a certain minimum occurrence (e.g. 3), then for each tag, it uses the web page DOM tree to find the tag's block and retrieve the most frequently used block. This most frequently used block is then marked as the data block and its data region/table name is retrieved with its schema.
2) Schema can be dynamically discovered for any given web page using our technique while with the WebOMiner, to add and learn about a new or re-structured B2C web page, the NFAs need to be refreshed so that it can recognize new features not previously in its structure. Our proposed system uses regular expression matching in the Java xpath parser through its methods such as compile(), evaluate() which will discover the most frequent structure block level tag (e.g., $< divclass = \text{“ ”} >$) in the Dom tree. The most frequent tags (such as tr tags) representing repeated pattern (records such as laptop product instances) indicate needed data blocks.
3) Our algorithm does not follow any hard matching techniques like tree alignment (such as recognizing zones and blocks of the DOM tree with series 1 and 2 observations) or NFA as done in [6]. Instead we summarize and observed the occurrence of tags that create similar block structure. The research challenge of developing the FSFinder is with finding a more dynamic automated approach in xpath parser, with the use of css div class names in the web page html files, and frequent pattern mining of relevant tag summaries to handle data heterogeneity in different B2C web sites which allows for future comparative mining.
4) WebOMiner_Simple is more accessible, extentdable and has the potential to be scalable to large websites because of its simple and effective technique.
5) It is highly automated and no manual efforts (such as description of sample training B2C pages) are required in the extraction process. We only have to give the system a product web page address and it will discover the database schema.

1.2 Related Work on Web Data Extractors

Researchers have worked on this problem for years, initially there were systems that were manual ([3], [7]). An Expert programmer has to manually observe the html source code of every web page and write code according to it such that it extracts required fields. This is a very resource consuming way although it yields highly accurate results, it is impractical for the purpose. Some systems tried to ease the task by using GUI where a user can select the required data while the system will do the rest of the work [8]. Supervised systems evolved where the algorithm learns from training examples and performs extraction task by using regular expression matching [10] and others like [11] had used Embedded catalog formalism on a hierarchical data extraction model. However, none of them was user independent or scalable. Data extraction had become semi supervised after IEPAD [2] in which PAT trees and multiple sequence alignments were used and the user had to select the required pattern out of extracted ones. But, some assumptions that were made were unrealistic and it was not fully automated. The process was almost automated by using techniques like partial tree alignment [14], ACME matching technique [5]. Many IE systems only perform record level extraction which is time consuming task that does not support non html web pages while html pages can be developed from several other technologies such as xml, ajax. Along with these, some of the existing systems have to scan the page multiple times which is resource consuming and a drawback. The WebOMiner [6] is a system that proposes a 4 phase architecture for the comparative mining of web

data. This system takes as input a web page populated with products to its first module the Crawler. They have developed a mini crawler algorithm that crawls the web page from internet and stores html files in a local repository. This is then given to their second module the cleaner, because according to [6], web data is ill formatted and before performing mining it is important to preprocess and clean the data to obtain well formed html files. They have used the Html_Cleaner 2.2 [12] to perform the data cleaning. All the comments were removed and tags were well structured after performing the pre-processing step. The output was given to the extractor module, it then constructs DOM tree from the cleaned html file and uses non-deterministic finite state automata to extract product tuples. Then, the tuples presented in array lists are stored in a database. Their architecture includes second level mining for future work, which implies to build a data warehouse serving for complex quering such as comparative historical and derived query mining. Their system performs matching with NFA for each product tuple which is a complex process that needs to be updated with restructuring of B2C web pages or emmergence of new web pages, also it is not yet able to handle long tag attributes. An example long tag attribute is:
src="/ScriptResource.axd?d=yUWPCPZyEuJXx3ZFmg
TnxgX_PUIOAaE3sD5dvOT0UArNJ-7jcDMMWg220GQi
97WCxi3NDXH7Bh_wyFkyH0fCRKHALWCBp2kc
QE_r_wiV3p_ZPD9YXmnEE-cci7LXLH0hDgsFG
IEEkqXN_hDdvbtV7OBldsZw1mQ5nXtfxgirstuTL4
TL0&t=2e2045e2".
To summarise, there is need for a system capable of using a realistic and uncomplicated scientific technique for this heavy web page data extraction task as it should be easily understood so the system can be more easily extendable, deployable, adaptable to other domains and more automated.

1.3 Paper Outline

Section 1 has introduced the topic of web data extraction and wrappers, related work and existing such systems, contributions of the proposed system. Section 2 presents the proposed WebOMiner_S system with all of its components and algorithms with a running example, Section 3 discusses the experimental evaluations of the system while conclusions and future work are discussed in section 4..

2. THE PROPOSED WEBOMINER_S FOR WEB SCHEMA EXTRACTION

In this section we present our proposed system, the WebOMiner_S, an automatic web data extraction system which extracts product tuples from B2C websites. Every B2C website (such as Walmart or Best Buy), showcases its products (such as laptops) arranged in blocks by fetching them from underlying databases. As in [6], a data region contains data blocks and both can be enclosed within one or more region/block level tags such as $< div >$, $< table >$, $< tr >$, $< span >$. A data region (representing a database schema) in DOM tree consists of a set of data blocks (representing data records or tuples). All data records (blocks) in a region (such as data region, advertisement region), in general, represent similar set of data and are contiguous in a data region. In the context of web content, a data block B_i contains usually heterogeneous data types such as text string, image-file, price as string (of type long) representing distinctive related instances. For example, a product data region records (blocks) can be described as a schema in a tuple such as:
Product (title: string, image: image-file, product number: integer, brand: string, price:real). Thus, finding tags that are repeated most frequently around a a data region in the html file can be used to identify the attributes of a table schema whose name is the data region node name. A continuous repeated pattern (indicating records of a product table) will be found in the html source code which reveals the database table schema of the main data block of the web page.

Also, all the products on the same web page are styled similarly using the cascading style sheets (CSS). The cascading style sheets help the web developers to lay out the information on the web page. It is the meta information on how the html product data elements should be displayed on the web page. A CSS can be created to define styles for the webpages, including the design, layout and variations in display. According to W3techs.com [13], nearly all (specifically, 93.3%) of B2C websites use external CSS to get the desired look for their webpages. An external CSS contains several styles each identified by a unique class name. The class attribute is referred in the html tag to which the preferred style has to be allocated. CSS will be defined first then later it can be assigned to any field by using the html tag attribute $< class >$. A sample CSS file with two styles indicated by two classes is shown below.
div.prodlook{
1in font-weight:bold;
1in margin-left:30px}
div.prodprice{
1infont: sans-serif;
1incolor:red;
1inbackground:green}
A CSS file with Two Styles
This CSS is then defined in the web page html source using the two following $< divclass >$ tags.
$< divclass = \text{“}prodlook'' > \ldots < /div >$
$< divclass = \text{“}prodprice'' > \ldots < /div >$
The style prodlook is used by referring to its class attribute in the div tag using class attribute.

Java Xpath parser [9] is a language for finding information in an xml file. It is used to traverse an xml document. XPath is a java API used to navigate through elements and attributes in an XML document. XPath contains a library of standard functions with which XPath uses path expressions to navigate in XML documents. It uses path expressions to select nodes or node-sets in an XML document. It has several methods such as compile() and evaluate() for extracting an entire block from the Dom tree given the parent node, and for checking the occurrence of the given expression in the Dom tree. An example summary of some Xpath functions are: the slash symbol / used to select the root node; // used to select nodes in the document from the current node; the symbol period . used to select the current node; .. used to select the parent of the current node; and @ used to select attributes. We can specify dynamic xpath expressions using variable name in the expression as "+variabelname+". XPath wildcards can be used to select unknown XML nodes. For example, $*$ matches any element node, @$*$ matches any attribute node, node() matches any node of any kind. Xpath compile() method compiles the path expression

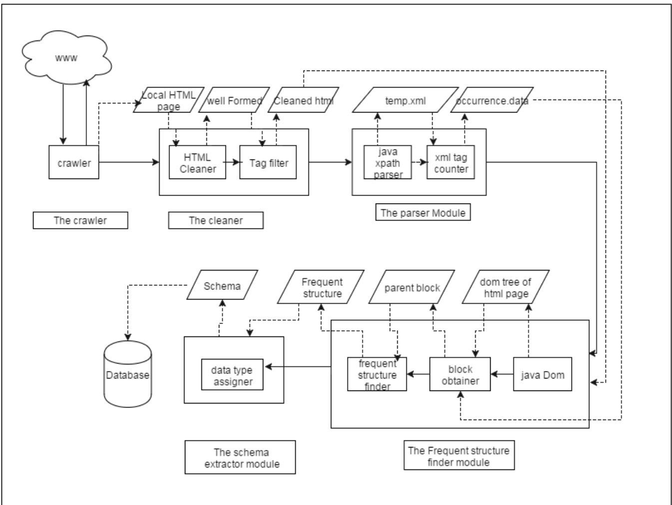

Figure 1: The WebOMiner WebOMiner_S Architecture

into an xpath object while Xpath evaluate() method returns specific data from xml file that matches the xpath expression.

After observing several B2C websites, we have noticed that all the products on a web page will have the same CSS for corresponding fields (such as product name, price). This means the same class will be assigned to all the product instances to the same fields (e.g., for each of the 8 laptop instances on web page of Figure 2, the same class is used for say field name) in the web page. We are making use of this observation in our proposal to look for and identify the block with most frequently repeated similar patterns for its attributes as the data region. Our proposed WebOMiner_S system follows the base architecture of the WebOMiner system [6]. Our WebOMiner_S system architecture given in Figure 1 has 5 modules namely, 1. The crawler, 2. The cleaner, 3. The Parser Module, 4. The Frequent Structure finder (FSFinder), 5. The schema Extractor. Each module is explained with its functionality and a running example as follows. The propositions 2.1 and 2.2 formalizing general B2C web site source html features and use of CSS class attributes to identify data blocks by our FSFinder algorithm are presented next.

PROPOSITION 2.1. *Identifying Blocks Dynamically: Blocks in an html file can be dynamically found by identifying all block tags with the same class name as in < div class = “classname′′ >. This is because all web sites using CSS files to describe the styles of block attributes have their source html code containing tags of the form < div class = “classname′′ >.* ■

PROPOSITION 2.2. *:Identifying Product Data Blocks: On a B2C website, the product block is the block with wht highest number of child nodes (tuples) and is the block that is repeated the highest number of time with the same CSS class attributes defined in < div class = “classname′′ > tag.* ■

The formal algorithm of the WebOMiner_S is given as Algorithm 1.

ALGORITHM 1. *(WebOMiner_S: Web Page Schema Extraction)*

Algorithm WebOMiner_S()
Input: *Web addresses of two or more product list pages (Weburls).*
Output: *Set of Table Schemas (S) for each Web document.*
Other: *Set of HTML files (WDHTMLFile) of web documents, WDHTMLFile#_ (set of clean HTML files), TempXMLFile, tagoccur file*
begin
(A) for each web address (Weburls)
begin
(1) WDHTMLFile = Call the Crawler() to extract webpage (Weburls) HTML into local directory.
(2) WDHTMLFile#_ = Call HTMLCLEANER-2.2 to clean-up HTML code
(3) Call the Parser() to first create temporary XML file of WDHTMLFile#_ as TempXMLFile, and next use the Xpath on TempXMLFile to summarize all tags's number of occurrence in a tagoccur file
(4) Call the Frequent_Structure() finder to first create a DOM tree with BuildDOMTree() using clean html file WDHTMLFile#_. Then, it uses the tagoccur file with the DOM tree to identify block and its attribute tags which has the most frequent similar repeating sequence of tags.
(5) Call Schema_Extractor() which uses the structure file and the DOM tree to associate each block attribute with data type and to output the discovered table schema in a file called schema.
end for
end

2.1 Components of the Proposed WebOMiner_S

Step 1: The Crawler Module
Our crawler module is responsible for extracting the webpage HTML source code of the given product web page from the World Wide Web (WWW). This module takes as its input the web page address, then extracts and stores its HTML source code in a file in the local computer directory. Our crawler module is similar to the WebOMiner's [6]. They (the WebOMiner system) have proposed a mini-crawler algorithm which crawls through the WWW to find targeted web page, streams entire web document including tags, texts and image contents and it then creates a mirror of original web document in the local computer directory. It dumps the comments from the html document. Figure 2 shows a sample input web page to our crawler, which is a product rich web page from the bestbuy.ca website.

Step 2: The Cleaner Module
The process of web data extraction requires preprocessing the input data which is an html page. Html is not a structured language and there can be missing tags, noise, comments etc. The sheer volume of raw html input source is also worth noting. For the example web page of Figure 2, the source html code consists of 144 pages of code starting with the tag < DOCTYPE/html > and ending with < /html >. This html source file has 4813 lines of codes with such tags as < script >, < noscript >, < forms >, < div >, < span >. So, preprocessing is necessary step to yield quality data. Like the WebOMiner system, we have used the HTML_CLEANER 2.2 [12] for performing the cleaning task. The output file will be a cleaned html file containing well formatted html structures with comments excluded. This file is given as input to the parser module to create temporary XML file and summarize the number of

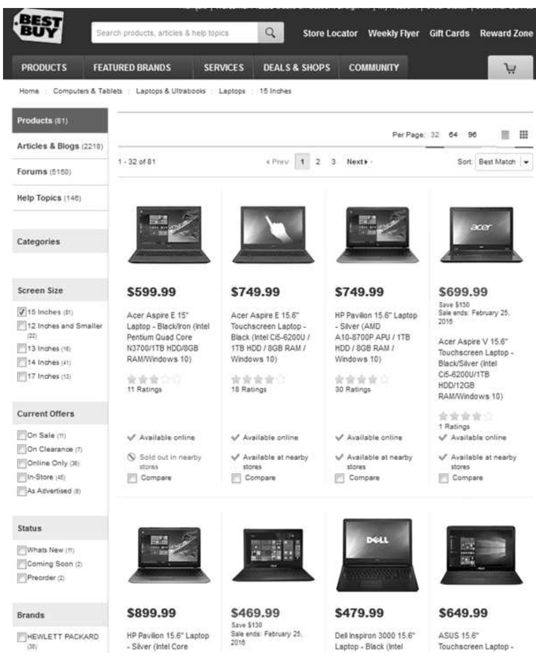

Figure 2: A Sample B2C Product Web Page to Extract its Schema

```
<li class="top-lvl menu-item parent-item parent-services">
        <a href="javascript:void(0)" class="link-top link- services">
                SERVICES</a>
        <ul class="sub-nav mega-menu menu-list">
            <li class="menu-item">
             <div class="services-content">
                <ul class="column-4">
                    <li class="menu-item first">
                      <a class="icon icon-gs" href="/en- CA/GeekSquad.aspx?
                      NVID =Services;Geek%20Squad;im;en">Geek Squad</a>
                      <p>From computer glitches to smartphone mishaps, Geek
                      Squad can save the day. Our Agents are ready and
                      waiting to set up, support, protect and repair.</p>
                      <a href="/en-CA/category/geek-squad/22042a.aspx?
                      NVID=Services;Geek%20Squad;im;en">
                      Learn more &gt;</a>
                   <li>
             </ul>
        </div>
       </li>
     </ul>
 </li>
</div>
        <div class="prod-saving">
                <span>Save </span>
                <span>$150</span>
        </div>
        <div class="prod-sale-ends">
                <span>Sale ends: </span>
```

Figure 3: Part of a Sample Clean HTML Source for Product Web Page

occurrences of tags. HtmlCleaner is open-source html parser written in Java. HTML found on the web is usually dirty, ill-formed and unsuitable for further processing. For any serious processing of such documents, it is necessary to first clean up the mess and bring the order to tags, attributes and ordinary text. For the given html document, HtmlCleaner reorders individual elements and produces well-formed html. By default, it follows similar rules that most web browsers use in order to create Document Object Model. However, users may provide custom tags and rule set for tag filtering and balancing. The HtmlCleaner inserts all the necessary closing tags and formats them to produce a clean html file as shown in the Figure 3.

Step 3: The Parser Module

There is need to convert our clean html file from step 2 to a temporary XML file containing only block level tags so that we can use the Xpath parser to evaluate the tags to retrieve their number of occurrences. The parser module takes as input the cleaned html file from the previous module of step 2. A temporary xml file (such as Figure 4) is created by recording all the blocks from the input html page. While creating, all the attributes in these block level tags like "id", "src" etc., were excluded and only the class attribute (such as $< divclass = \text{“ ''} >$) are retained as they hold the information regarding which particular class of CSS is being used by this particular block. Figure 4 shows the temporary xml file for the same page from the previous modules, where all the block tags were recorded with only class attributes. This entire document is parsed using Xpath parser ([9]) to check if any blocks were repeated with the same class name. Thus, the summary of found repeated tags (those occurring at least 3 times since it is uncommon to have B2C web sites listing less than 3 products) along with the number of occurrences are stored into a tag occurrence file as shown in Figure 5. The number of pages of xml code obtained after going through the parser reduces the original example html of size 144 pages to only 6 pages of xml code used for further processing.

Step 4: The Frequent Structure Finder (FSFinder) Module:

This modules initially creates a Dom tree for the cleaned html file from the second module. Figure 6 shows the part of Dom file constructed from the running example. FSFinder module takes as its input (1) tag occurrence file (Figure 5), (2) the clean html file (Figure 3). It builds Dom tree (Figure 6) using the clean html file. Then, for every block level tag with a class attribute such as $< divclass = \text{“ ''} >$, such as $< divclass = \text{“}proddetails'' >$ with 17 occurrences, the FSFinder creates a dynamic path expression for this tag. The xpath parser then searches the entire Dom tree for this expression using its methods compile() and evaluate() and returns results in a node list variable called struct_list. This struct_list now is the live reference to those particular nodes in the Dom tree, such as every occurrence of the node $< divclass = \text{“}proddetails'' >$ in the Dom tree is being referred from this node list. Now using this node list we can access the complete block using methods like getchildnode() and getlength(), as well as all the child node tags to this parent block. Also, the Nodelist object represents an ordered list of nodes. The nodes in the Nodelist can be accessed through their index number (starting from 0). The Nodelist keeps itself up-to-date. If an element is deleted or added in the Nodelist or XML document, the list

```
<divclass="department-headline">
<divclass="ui-tabcontrol">
<divclass="ui-tabcontrol-tabs">
<divclass="ui-tabcontrol-tabselected">
<divclass="ui-tabcontrol-tab">
<divclass="clear">
<divclass="ui-tabcontrol-contents">
<divclass="ui-tabcontrol-contentselected">
<divclass="product-containerclear-fixl1-across">
<divrenderemptytext="false">
<divclass="productlast">
<divclass="theme-c">
<divclass="theme-product-headlineclear-fix">
<tableclass="theme-headline-inner-table">
<tdclass="theme-headline-inner-text">
<divclass="inner-product-wrapper">
<divclass="product-content-wrapperclear-fix">
<divclass="theme-left-column">
<divclass="l1-across-swatches">
<divclass="theme-right-column">
<divclass="release-datebtm-space">
<divclass="l1-across-pricepill-wrapperclear-fixbtm-space">
<divclass="pricepill">
<divclass="rebate-top">
<divclass="pricepillred">
<divclass="btm-spacevalid-date">
<divclass="priceblock">
```

Figure 4: Part of Parsed XML File of Input Web Page

```
<divclass="a9prodwrap"> 12

<divclass="prodoffer"> 17

<divclass="prodtitle"> 17

<divclass="prodimage">25

<divclass="proddetails"> 17

<divclass="contentno-bottom-pad">

<divrenderemptytext="false"> 3

<divclass="productlast"> 5

<divclass="theme-product-headlineclear-fix"> 7

<divclass="inner-product-wrapper"> 9

<divclass="product-content-wrapperclear-fix"> 9

<divclass="pricepill"> 6

<divclass="rebate-top"> 6

<divclass="pricepillred"> 6

<divclass="more-like-this"align="center"> 8

<divclass="fieldbox-ftr-br-col"> 3
```

Figure 5: Summary of Tag Occurrences of Input Web Page

Table 1: The Table Schema of the Input Web Page

$< divclass = \text{“}a9\ prodWrap'' >$
$\quad < divclass = \text{“}prodImage'' >$
$\quad\quad < divclass = \text{“}prod - image'' > \ldots < /div >$
$\quad < /div >$
$\quad < divclass = \text{“}prodDetails'' > \ldots < /div >$
$\quad < divclass = \text{“}quickview'' > \ldots < /div >$
$< /div >$
$< divclass = \text{“}clear'' > \ldots < /div >$
$< /div >$

is automatically updated. In a Nodelist, the nodes are returned in the order which they are specified in the XML document (which corresponds to pre-order traversal (visit node, visit left subtree, visit right subtree) of the DOM subtree. Nodelists are live references to actual DOM elements. The number of children and length of the struct_list are retrieved using getchildnodes() and getlength() respectively to realize the block that has maximum number of children and which is repeated most number of times. This process is repeated till the end of the file. As we complete, we have the parent nodes to the product blocks in the Dom tree in the node list data tuples. Using the first parent node indicated by index 0 in the struct_list, we retrieve the complete structure from the Dom tree by reading each child node into a Nodelist in the pre order format into FRS. Therefore the FRS has the most frequently repeated structure with more number of Childs in the dom tree. This structure is further used to retrieve database schema. In our example, the structure that is repeated most number of times with maximum child nodes is shown below and in Table 1.
$< divclass = \text{“}a9\ prodWrap'' > \rightarrow < divclass =$
$\text{“}prodImage'' > \rightarrow < divclass = \text{“}prod - image'' >$
$< /div > \rightarrow < /div > \rightarrow < divclass = \text{“}prodDetails'' >$
$< /div > \rightarrow < divclass = \text{“}quickview'' > \rightarrow$
$< /div > \rightarrow < /div > \rightarrow < divclass = \text{“}clear'' >$
$< /div > \rightarrow < /div >$
Thus, through the FSFinder algorithm, we have obtained the structure (FRS) and from this structure, we can extract the underlying table schema of the web page. Also, note that the FSFinder algorithm has in the Nodelist data_tuple, all the attribute values of the product data block which are the tuples or records of the web page for the retrieved schema. The formal algorithm for the frequent structure finder (FSFinder) is given as Algorithm 2.

Algorithm 2. *(The File Structure Finder Algorithm)*

Algorithm FSFinder()
Input: *occurrence data file (occur), cleaned html file (clean)*
Output: *Nodelist frequent_structure (FRS)=null*
(for product blocks names), data_tuples
(for product blocks values)
other: *Domtree: file, product_block:string,*
FINAL_nodelength = 0, no_of_childs = 0, final_childs = 0
xmlnodetag: string (for complete block node),
xpathExpression_expr: string, path : string
Nodelist prod_block (for block node children)
Nodelist Tagstructure, struct_list, data_tuple
xmlnodetag_IDCount = 0, nodes_length: integer
Packages: *Java DOM, Java Xpath, Java transformer*
begin
1. Domtree = Call DocumentBuilderFactory(clean)

```
<div class="a9 prodWrap">
        <div class="prodOffer">
         <h4 class="prodTitle">
           <a id="ctl00_CP_ctl00_C3_ctl00_LT_SL_ctl01_ctl00_
           CalltoAction" href="http://www.bestbuy.ca/en-CA/amd-
           a-series.aspx"> Free downloadable copy of Battlefield 3
           _ and Dirt 3 with HP AMD A8 APU laptops
          </a>
        </h4>
        </div>
</div>
<div class="prodDetails">
        <div class="prodImage std-bottommargin">
        <div class="prod-image">
          <a id="ctl00_CP_ctl00_C3_ctl00_LT_SL_ctl01_
          ctl00_PI_HI" href="http://www.bestbuy.ca/en-CA/
          amd-a-series.aspx">
         <img src="/multimedia/Products/100x100/101/10181/10181707.jpg"
         alt="Battlefield 3 (PC) - English" class="img100x100" />
        </a>
        </div>
        </div>
        <p class="details">
        <a id="ctl00_CP_ctl00_C3_ctl00_LT_SL_ctl01_ctl00_Offer"
        href="http://www.bestbuy.ca/en-CA/amd-a-series.aspx">Shop &gt;
        </a>
        </p>
</div>
```

Figure 6: Part of DOM Tree of the Input Web Page

```
2. Create object xpath for xpathfactory() to use its
methods to retrieve all nodes that match the path
expression from Dom tree.
3. for each xml tag in tag occurrence file occur,
// find the number of occurrences and the number
// of children so we can find the tag with most
// number of children which is the most repeated tags.
    3.1 Retrieved xml is stored into xmlnodetag[1]
    3.2 path = "//div[@class='"+xmlnodetag[1]+"']";
    // The class attribute in xmlnodetag is assigned to
    // the variable dynamically in each loop
    3.3 xpathExpression_expr =xpath.compile(path)
    3.4 struct_list= xpath.evaluate(Domtree,
    XPathConstants.NODESET)
    // This retrieves all the nodes that match the
    // path expression from the DOM tree
    3.5 node_length = struct_list.getlength()
    3.6 no_of_childs= struct_list(0).
    getchildnodes().getlength()
    3.7 If (node_length >= FINAL_nodelength and
    no-of-childs > final_childs)
        3.7.1 product_block = xmlnodetag[1]
        3.7.2 FINAL_nodelength=node_length
        3.7.3 final_childs = no_of_childs
        3.7.4 data_tuple=struct_list
end for
4. if product_block has child
5. for each child node in product block
    begin
        5.1 FRS = FRS append product_block.child
        // this appends child node of product bloct to FRS
        5.2 Read next node
    end for
6. Return Nodelist data_tuple
end
```

Table 2: The Extracted Table Schema of the Web Page

Discovered schema for a single page of bestbuyWebsite is below:
Product(prod-image: String, img300x300: image, prodDetails: String, prodTitle: String, prodPrice: String, priceblock: String, pricetitle: String, shop-now: String, customer-rating: String, rating-title: String, rating-stars: String)

Step 5: The Extractor Module
This module performs the task of extracting both the table schema and the product tuples from the frequent structure (sequence of tags) and the data_tuple objects returned by the FSFinder algorithm. It converts the schema of the database table retrieved by the FSFinder algorithm to a data warehouse schema by appending an integration attribute called "storeid", and a history attribute called "time". Those two attributes will enable integration of tuples from different web sites at different timestamps for comparative and historical querying. A product tuple P is a product with attributes a_1, a_2, a_3, ..., a_n. This module takes as input both the frequent structure pattern and the data_tuple structure generated by the previous module. It is given to a function that assigns corresponding datatypes based on the tags to the frequent structure pattern for the table schema. If it is an $< img >$ tag the data type image will be assigned, if it is any of $< p >$, $< span >$, $< li >$, $< a >$ tags, data type "string" will be assigned, etc. Finally, the data warehouse schema is discovered from the web page. This can be used to create and update a database table and further creating a data warehouse when an integrative attribute "storeid" and a historical attribute "time" are appended to the table schema. Having a data warehouse schema allows for comparative mining and historical querying. Table 2 shows the table schema extracted from the running example. Once the database or warehouse schema is defined, the actual data tuple instances can be extracted from the nodelist data_tuples which holds all the product blocks with their parent nodes. From the dom tree, each parent node is checked if it has child nodes and values are updated into the database/warehouse table.

2.2 Example Application of the WebOMiner_S for Web Schema Extraction

EXAMPLE 1:
Given a product list web page (such as a laptop page from bestbuy.ca) of one or more retail B2C web sites such as that with 8 products as shown in Figure 2, using the WebOMiner_S system, extract all types of information like: (i) Those related to data records such as product image, product brand, product id, short description, product price. In our current example, a laptop page from best buy, the information to be extracted will be laptop image, name, brand, screen size, screen type, color, CPU info, HD info, RAM info, OS, Ratings, Availability. The extracted information will be stored in the database/warehouse for future comparative mining and querying.

SOLUTION 1: The proposed WebOMiner_S system given as algorithm 1 implemented in JAVA runs in both Unix and Windows based environment (with NetBeans or Eclipse) to store the extracted data from the web page in a DBMS like Oracle or MySQL tables. The directory where the system is run requires having the following files: WEBOMINERS.jar, htmlcleaner-2.2.jar, ojdbc6.jar and the downloaded and cleaned web page to be extracted which is stored as cleanHTML#_.html. The algorithm would download the single pages when provided the link and the cleaned page clean-HTML#_.html was obtained after running steps 1 and 2 on the web page of Figure 2. Note that all cleaned web pages to be extracted are currently stored in
cleanHTML#_.html (e.g., bestbuy_.html). The Unix command for initiating the extraction of the cleanHTML#_.html page is given below: java -cp WEBOMINERS.jar:htmlcleaner-2.2.jar:ojdbc6.jar Main.class. This command has the effect of going through steps 3 to 5 of the WebOMiner_S algorithm using the input file cleanHTML#_.html such as given in Figure 3. The third line of the WebOMiner_S algorithm calls the parser() method to parse the cleaned html document of Figure 3. For each line in this clean html file, the parser looks for and retains the tags $< div >$, $< table >$, $< tr >$, $< td >$ in the temporary xmlfile, as these are the tags that are used by all the web sites to embed information in the webpage and also to define a block section in the html document. Only all of such block tags are saved into a temporary xml file (4) by the parser. Note that only parts of the long 144 page original html source code of the example web page are displayed in these figures. For example, the input clean file of Figure 3 will yield the output xml file the following tags:
$< divclass = \text{``}services - content'' > \ldots < /div >< /div >$
Next, for each line in the temporary xml file, the parser algorithm returns the number of occurrences of the given line in the entire temporary xml. For example, given the xml file input of Figure 4, the parser and the tag line$< divclass = \text{``}department - headline'' >$, it searches the whole file to find the number of lines containing this tag and inserts it in the output tag occurrence file shown as Figure 5. Step 4 of the main WebOMiner_S algorithm, calls the FSFinder() which creates the DOM tree (Figure 6) using clean html file of Figure 3 and retrieves a structure (a sequence of block tags) with their data tuples, which has occurred most frequently in the DOM tree also given the input tag occurrence file of Figure 5. For the running example the structure found is as shown in Table 2. In line 3 of FSFinder algorithm, the loop iterates to read a line from the tag occurrence file and the first line read is " $< divclass = \text{``}a9prodwrap''/ >''$. Since this line contains a class attribute, so this line is processed. First, it extracts the value of class attributes and creates a dynamic "path" string object with the value of class (path="//div[@class=a9prodwrap]"). This "path" variable passes to compile method "xpath" object "expr". The evaluate() method of "XpathExpression" extracts all the nodes from the DOM tree which is matched with "class=a9prodwrap" embedded with "div" tag and saves this result set to Nodelist object struct_list. Now, the struct_list has reference to every occurrence of the given tag in the Dom tree. To know the number of occurrences, the length of the node list is extracted using getLength() and this value is saved to variable "numberofnodes". There is an integer variable called "finalofNodes", which keeps record of maximum number of nodes by comparing with "numberofnodes". Initially, the value of "finalnoofnode" is set to "0". During the first iteration, the value of the number of nodes is 12 and "finalnoofnodes" is 0 and noofchildnodes is 18, finalchlds is 0. Since the value is greater than and thus satisfies the condition, control will enter into IF block and the values of finalnoofnodes and finachilds get updated along with the product_block which contains the parent to the product block. Since this set of nodes had satisfied our condition, we consider them to be our future product blocks and store this node list into another node list called data_tuples. The loop iterator reads next line which is $< divclass = prodoffer >''$. Since this line contains a class attribute so this line is processed. First, it extracts the value of class attributes and creates a "path" string object with the value of class (path="//div[@class="prodoffer"]"). This "path" variable passes to compile method "xpath" object "expr". The evaluate() method of "XpathExpression" extract all the nodes from the DOM tree which is matched with "class=prodoffer" embedded with "div" tag and save this result set to Nodelist object. Now the noofnodes is 17, noofchild is 5 since this does not satisfy the condition, the values of variables remain unchanged. Thus, this process carries until the tag occurrence file data gets to the end. After the loop ends, the variable product_block will have the parent of the block that repeated the most number of times with the maximum child elements which is the block of focus. Now, this block is read recursively in the lines of 5 and stored into a nodelist tagstructure. Finally, we have the structure that is most repeated frequently all over the dom tree which is later passed to the schema extractor module to discover database schema of the web page and data_tuples which has the parent nodes to all the product block. Now that we have the most frequently repeated structure all over the web page, it is used to derive the database schema of the web page. The main algorithm encounters step 5 where it calls the schema_extractor() method to extract schema from the structure found. Each tag from the structure is passed to this method, it then checks if the tag is $< img >$, it assigns blob or image as data type. If it is a text tag, it assigns varchar as the data type as shown in Table 1. The nodelist data_tuples consist of all the product blocks with their parent nodes. From the dom tree, each parent node is checked if it has child nodes and values are updated into the database table. This data further can be used to create a data warehouse for comparative mining and historical querying. In our own case, we append the data warehouse integration attribute "storeid" of type string and the historical attribute "time" of type string.

3. EXPERIMENTAL EVALUATIONS

The integration of all the modules are built in together to form a software bundle which accepts the URL of the web site. The software processes the web content and generates the schema of the products. The experiment is done with 4 different web sites (bestbuy.com, futureshop.ca, canadiantire.com, walmart.ca) for empirical evaluation of our system using different page structures. Our system is implemented in Java programming language, run on 64-bit Windows 7 home operating system with Intel Due Core 2.26 GHz, 4.00 GB RAM hp machine for empirical evaluation of our schema extractor system. We use the standard precision and recall measures to evaluate the results of our extraction

Table 3: Results of Webominer_S extraction of schema from web pages

Web page	Actual Schema			Extracted Schema			
	Fields count	String fields	Image fields	Correct	Wrong	Missing	Irrelevant
bestbuy.ca	11	9	2	10	1	0	1
futureshop.ca	12	10	2	12	0	0	6
canadiantire.ca	17	14	3	15	2	0	5
walart.ca	15	12	3	12	1	0	2

system. Precision is measured as the average in percentage for the number of correct data retrieved divided by the total number of data retrieved by the system. Recall is measured as the average in percentage for the total number of correct data retrieved divided by the total number of existing data in the web document. While the precision for the schema extraction achieved by the system using the 4 web pages is 80%, the recall achieved is 90%. The precision includes irrelevant fields extracted even though all necessary product fields on the web page are extracted. Results of the retrieval by our schema extractor system are tabulated in Table 3:

The purpose of our experiment is to measure the performance of the WebOMiner_S schema extractor system. Table 3 shows small scale experimental results as performance measure for our schema extraction system. We have taken one page per web site for experiment and the numbers in the columns show different types of data in those pages. The "Fields count" column shows the total number of attributes in the schema of the product block for each of the downloaded pages. For those pages schema extractor system is able to identify schema correctly. Very few wrong data record identification is observed and it makes sense because our system is not based on use of sample training pages. It misses no actual information on the page because it extracts the data from web pages from different websites. There are 14 irrelevant attributes which are generated in the extraction of schema. We observed the reason for irrelevant attributes. All of those irrelevant attributes extracted are in the List type data records which mixes objects of different types in a data tuple. Our definition of List data tuple is a set of $< text >$ and there should be at least 3-pairs in the tuple to be qualified as List tuple. This means that those list consisting of both $< image >$ and $< text >$ do not satisfy this criteria and will not be recognised. This remains area for future work.

4. CONCLUSIONS AND FUTURE WORK

There is need for a system capable of performing deeper knowledge discovery consisting of comparative analysis of such product features as prices, answering historical and derived queries about products and other data on web pages. This paper proposes an approach for solving this complex data extraction problem using JAVA Xpath parser methods with web page DOM tree tag node frequent structure count summarization and mining. The proposed WebOMiner_S system summarizes the number of times each tag in a web page DOM tree has occurred. Then, it uses this data to discover the most frequently occurring tags since those most likely would belong to the product blocks. Then, it uses these frequent tags to go back to the DOM tree to retrieve the parent tag node (corresponding to the table name) and all its children tags nodes which will correspond to the table attributes. This approach advances the more complex non-deterministic finite state automata employed by an earlier system the WebOminer, which also requires sample training B2C web pages. The proposed WebOMiner system discovers the schemas of the B2C web pages without the need for training web pages and building NFA for different content types. This has the potential to make the system more extendable and accessible to users. We have demonstrated that this first implementation phase of this system is effective for extracting web schema and contents and storing them in database tables for querying and mining. Future improvement on the proposed system includes: the crawler module needs to create the functionality for more automatic selection of the targeted documents from the web, cleaner module needs to handle long tag attributes. Future extensions of the system are ongoing research. We feel there is plenty of room for improvement and to open new thread. With this method the data types are not yet well defined and the analysis of the child tag would give the data type close to the original data. Current system does not handle "on click" contents on the web sites which are dynamically generated on user's request and loaded with tools such as AJAX, jQuery functions like jQuery.load() or content loaded by scrolling. Future work should extend the system to dynamic contents. Our FS-Finder can be extended such that frequent structures in the dynamic web page is detected through their usage external css with xpath parser.

5. ACKNOWLEDGMENTS

This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0194134) and a University of Windsor grant.

6. REFERENCES

[1] E. Annoni and C. Ezeife. Modeling web documents as objects for automatic web content extraction-object-oriented web data model. In *ICEIS (1)*, pages 91–100, 2009.

[2] C.-H. Chang and S.-C. Lui. Iepad: information extraction based on pattern discovery. In *Proceedings of the 10th international conference on World Wide Web*, pages 681–688. ACM, 2001.

[3] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The tsimmis project: Integration of heterogenous information sources. In *Proceeding of IPSIâĂŹ94, Japan*, March 1994.

[4] W. Consortium et al. Html5. https://www.w3.org/TR/html5/, 2013.

[5] V. Crescenzi, G. Mecca, P. Merialdo, et al. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, volume 1, pages 109–118, 2001.

[6] C. Ezeife and T. Mutsuddy. Towards comparative mining of web document objects with nfa: Webominer system. *International Journal of Data Warehousing and Mining (IJDWM)*, 8(4):1–21, 2012.

[7] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The tsimmis approach to mediation: Data models and languages. *Journal of intelligent information systems*, 8(2):117–132, 1997.

[8] J. Hammer, J. McHugh, and H. Garcia-Molina. Semistructured data: The tsimmis experience. In *Proceedings of the of the 1st East-European Conference on*, pages 1–7, 1997.

[9] java.com. Learn about java technology. https://www.java.com/en/about, 2016.

[10] A. H. Laender, B. Ribeiro-Neto, and A. S. da Silva. Debye–data extraction by example. *Data & Knowledge Engineering*, 40(2):121–154, 2002.

[11] I. Muslea, S. Minton, and C. A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):93–114, 2001.

[12] Sourceforge.net. Htmlcleaner web page: Transforms html to well-formed xml. http://htmlcleaner.sourceforge.net/index.php, 2015.

[13] W3Techs. Web technologies surveys: Usage of css for websites. https://w3techs.com/technologies/details/ce-css/all/all, 2016.

[14] 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. ACM, 2005.