2013

Comparative Mining of Multiple Web Data Source Contents with Object Oriented Model

Yanal Alahmad

Comparative Mining of Multiple Web Data Source Contents with Object Oriented Model

By

Yanal Alahmad

A Thesis

Submitted to the Faculty of Graduate Studies through the School of Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2012

Comparative Mining of Multiple Web Data Source Contents with Object Oriented Model

By

Yanal Alahmad

APPROVED BY:

Dr. Abdulkadir Hussein, External Reader

Department of Mathematics and Statistics

Dr. Xiaobu Yuan, Internal Reader

School of Computer Science

Dr. Christie I. Ezeife, Advisor

School of Computer Science

Dr. Alioune Ngom, Chair

School of Computer Science

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

Web contents usually contain different types of data which are embedded under different complex structures. Existing approaches for extracting data contents from the web are manual wrappers, supervised wrapper induction, or automatic data extraction. The WebOminer system is an automatic extraction system that attempts to extract diverse heterogeneous web contents by modeling web sites as object oriented schemas. The goal is to generate and integrate various web site object schemas for deeper comparative querying of historical and derived contents of Business to Customer (B2C) such as BestBuy and Future Shop. The current WebOMiner system generates and extracts from only one product list page (e.g., computer page) of B2C web sites and still needs to generate and extract from a more comprehensive web site object schemas (e.g., those of Computer, Laptop and Desktop products). The current WebOMiner system does not yet handle historical aspects of data objects from different web pages.

This thesis extends and advances the WebOMiner system to automatically generate a more comprehensive web site object schema, extract and mine structured web contents from different web pages based on objects' patterns similarity matching, and stores the extracted objects in historical object-oriented data warehouse. Approaches to be used include similarity matching of DOM tree tag nodes for identifying data blocks and data regions, automatic Non-Deterministic and Deterministic Finite Automata (NFA and DFA) for generating web site object schemas and content extraction, which contain similar data objects. Experimental results show that our system is effective and able to extract and mine structured data tuples from different web websites with 79% recall and 100% precision. The average execution time of our system is 21.8 seconds.

Keywords: Web content mining, Object-Oriented, Web data extraction, Wrapper induction, Frequent Objects Mining, Data warehouse, DOM-Tree.

ACKNOWLEDGEMENT

I would like to give my sincere appreciation to all of the people who have helped me throughout my education. I express my heartfelt gratitude to my parents for their support throughout my graduate studies.

I am very grateful to my supervisor, Dr. Christie Ezeife for her continuous support throughout my graduate study. She always guided me and encouraged me throughout the process of this research work, taking time to read all my thesis updates. Really, I have got financial support and research assistantship positions from Dr. Ezeife which helped me to proceed with my study.

I would also like to thank my external reader, Dr. Abdulkadir Hussein, Dr. Xiaobu Yuan, my internal reader and my thesis committee chair, Dr. Alioune Ngom for making time to be in my thesis committee, reading the thesis and providing valuable input. I appreciate all your valuable suggestions and the time, which have helped improve the quality of this thesis.

At last, I would express my appreciations to all my friends and colleagues, especially Tamanna Mumu, Gunjan Soni, and Sabbir Ahmed for their help and support. Thank you all!

DEDICATION

This thesis is dedicated to my parents, my mother Hind Alzoubi and my father Nawaf Alahmad.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 - Introduction

Number of web pages is growing very fast every day and World Wide Web (WWW) now represents a huge repository of data source in the world. Web coverage of information is very wide and diverse, where users find some difficulties to retrieve all the information they want. So, web data analysis and web content mining have become very important research area. Web content mining aims to extract useful knowledge from the contents of web and conclude future decision based on this knowledge. For example, web content mining can extract potential useful information about products or individual item from different web sites such as prices, titles, products series, etc. Web contents are heterogeneous in nature and can be in different formats, e.g., structured tables, texts, images, links, multimedia data, etc. So far there is no complete automatic extraction model catches the full diversity of web contents (Annoni and Ezeife, 2009).

There are three main classes of data extraction from web. The first class is called manual extraction, where a user or developer manually labels targeted items inside a web page and writes the extraction rules to extract such items. The manual approach suffers from many of problems, it is considered time consuming, requires a lot of human efforts to write extraction rules and update them. The second class is called wrapper induction, where a set of manually labeled pages are given and machine-learning techniques are applied to identify specific patterns and build extraction rules from large initial training web pages. The extraction rules are applied for further manipulation and extraction of data from subsequent pages that contain important information similar to those pages in training collections. Wrapper induction suffers from several problems, where manual labeling is still labor intensive and time consuming. In addition, wrapper needs regular maintenance by experts to accommodate the frequent changes and updates of websites to keep the extraction rules valid. The third class is automatic extraction, where a set of training pages are given and the extraction rules are built automatically. Automatic extraction system is able to extract web contents even if only one training page is given. Many of researchers consider current automatic web content extraction methods as inaccurate and make many assumptions about web pages which need to be extracted (Zhai and Liu 2005).

Many of the important information on the web are contained in regularly structured format such as list of online electronic products objects. Such objects represent structured database records generated from underlying database of website and displayed in web page in a regular structured format. Data objects are valuable and very important because they represent the main theme of their websites. Often a list of such continuous objects represents a list of similar items, e.g., list of products, books, services, etc. Mining data objects is very useful because it allows different information from different web pages to be integrated together in one database, which add more web services like shopping comparisons, e-commerce, and web search. Web Sources allow access to deep web and underlying database in HTML or semi-structured format, which makes it difficult task on any software to extract data objects and their related attributes from web pages. Annoni and Ezeife (2009) propose a new model called *OWebMiner* to represent web contents as objects. They encapsulate each web content type as object-oriented class to catch the heterogeneous contents together based on the page structure. Annoni and Ezeife (2009) depend on the visual coordination of web page contents to define the data block and data region that contains similar data objects which complicates the extraction process. They also depend only on the HTML tags without attributes in the comparisons of data objects which are sometimes not enough to recognise what type of object is. Also they do not provide any result to show the output of extracting web data objects from any list product web page to do a simple query from that page. Mutsuddy (2010) developed the work by (Annoni and Ezeife, 2009) to build *WebOMiner* system to wrap web contents as objects and then extract these objects and store them in relational database. WebOMiner system is a hierarchal system that represents web page as a DOM tree to facilitate the extraction process based on the proposed definition of block-level and non block-level of data block by (Mutsuddy 2010; Ezeife and Mutsuddy, 2013). The authors propose a Non Deterministic Finite state Automation (NFA) approach to define the structure of each extracted data block. WebOMiner system suffers from some shortcomings. It defines the NFA structures of the data tuples based on the wide authors' observations of the B2C websites product list data schema (e.g., product <title, image, price, brand>) which makes the extraction process not fully automatic and limits the performance of the proposed system. The proposed extraction process does not guarantee unique identification of the complete data objects which are related to one data block. The current existing WebOMiner is able to extract data objects from only one list product web page of B2C website, and has not been tested to extract the objects from different list product web pages. This means that the generated schema of a B2C web site has only one product class. In addition, the authors do not introduce any clear automatic database structure to store the extracted data objects. Besides that, the WebOMiner is not yet a historical system that can answer such queries that are related to historical data, such as *what was the price of Dell CORE i5 and 6 GB RAM Laptop in the August of 2011?*. Harunorrashid (2012) advanced WebOMiner to a new version called WebOMiner-2, where the author try to automate the process of building the NFA structure of data block types based on the generated regular expression of repeated object contents. The WebOMiner-2 still is not fully automatic and suffers from some shortcomings. It builds the structure of only one product data block. Also it extracts the data object contents from only one list product web page, in addition the proposed database structure is relational database (RDB) and it is neither historical nor object-oriented. Moreover, WebOMiner-2 does not provide any GUI to allow the end user to do 'Comparative Shopping' between extracted product items and answer queries related to them. Zhang (2011) proposed object-oriented class model, database schema, and object-oriented join (OOJoin) method to join superclass and subclass tables by matching their types and their hierarchical relationships, then mine hierarchical frequent patterns (MineHFPs) from multiple integrated databases using the extended TidFP technique (Ezeife and Zhang, 2009).

This thesis studies the idea of modeling web contents as data objects and extends the current existing WebOMiner and WebOMiner-2 systems through modifying their techniques and algorithms. The new modifications include modifying the cleaner module to do better cleaning of the extracted web page by removing extra comments, scripts, style sheets, and metadata which are contained inside the html code of the page. Define a new extraction algorithm based on the DOM tree tag nodes similarity matching technique to identify similar data blocks; the new algorithm guarantees extraction of all the matched data tuples and their related attributes to distinguish each data tuple individually. We also propose to fully automate the process of defining the NFA structure for more than one class of tuple type and generate the complete schemas of different websites. The new modified version of NFA generator algorithm builds frequent object tree (FO-Tree) to compute the frequency of data attributes and preserve the sequence order and relationship between the attributes, then the FO-Tree is mined to generate the frequent pattern of data blocks which represent the structure of website schema. We also propose to build historical object-oriented data warehouse (OODWH) schema to integrate and store the extracted data tuples from different web pages in the domain of B2C websites. The OODWH schema is generated by matching different NFA schemas for different websites sources to define the common attributes of the products and build the fact and dimension tables for the integration purposes. The proposed data warehouse will be able to integrate the different structure class of different products objects from different websites, and store the historical information about these products. The rest of this chapter is organised as section 1.1 introduces web mining and its categories; section 1.2 introduces object-oriented data warehouse model; section 1.3 introduces types of extracted web pages; section 1.4 thesis problem statement and section 1.5 thesis contributions; and section 1.6 outline of the thesis proposal.

1.1 Web Mining

Web mining is one of the data mining applications techniques to analyze and extract relevant knowledge from the web to make future decisions. For example, Internet Service Provider (ISP) Company uses web mining techniques to analyze users' web browsing patterns to identify users' web interests and their needs. Etzioni (1996) consider web mining as a data mining technique to extract knowledge from World Wide Web pages and services. Web is very large, diverse, dynamic, and mostly unstructured data storage, it raises the difficulty to deal with the information from different perspectives. How the users would get the relevant documents they want from search results with fast response time. The web service provider needs to monitor users' web usage to identify their interests. Business analysts need to identify users' needs to build suited websites to attract customers. All of them need techniques and methods to facilitate the extraction of web contents, and conclude the appropriate knowledge in easy and accurate way. For all the previous reasons web mining became very active and important research area. Borges and Levene (1999) classify web mining into three categorizations: web content mining, web structure mining, and web usage mining. Web contents are the primary information of web document, which usually include different types of data such as texts, images, hyperlinks. Kosala and Blockeel (2000) define web structures, the way how web contents are represented. Web usage is the history of user's visits of web pages sorted in chronological order in web log files. There are other web contents not considered important as primary information of web page. This unwanted web contents are called noise information and should be cleaned before mining web contents process begins (Gupta et al., 2005; Li and Ezeife, 2006).

1.1.1 Web Structure Mining

Web structure mining is a tool to define the structural relationship between web pages linked together by information or direct internal hyperlinks (Madria et al., 1999; Kosala and Blockeel, 2000). Hyperlinks are used in web pages to navigate from one page to other pages. Web content mining focuses on document level (inner-page) structure, while web structure mining focuses on hyperlink level (inter-page) to discover the model underlying the link structure of web page. The link connection between web pages is very important information, and can be used to filter web documents and rank them. The link structure model of web contains important information such as the relationships between websites, which can help in filtering and ranking web pages for search engines based on the importance of web pages. A link from page *A* to page *B* is a recommendation of importance of page *B* from author of page *A*. Set of connected web pages by internal hyperlinks can be represented as web structure graph as shown in figure 1, where each node represents a web page, and the edge represents the hyperlink between two web pages.

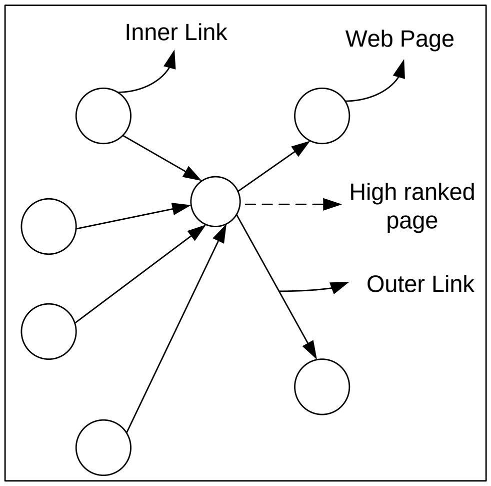


Figure 1: Web structure graph

Web structure graphs help to give a sign for the popularity of a website through computation of the in-degree and out-degree for each web page. In-degree of a web page *A* represents the number of web pages that have a direct link to *A*. The out-degree represents the number of internal links inside page *A* that refer to other web pages. As shown in figure 1 that the web page which is referred by label 'High ranked page' has the highest in-degree value which equals to four and out-degree equals to two. This means that this web page is important compare to other web pages.

1.1.2 Web Usage Mining

Srivastava et al., (2000) defines web usage mining as an application of data mining to extract knowledge from web usage log information, in order to understand and better serve the needs of web-based applications. Generally, web usage and users' visits to different web pages are stored in historical order in web log files. Log file can be: server log, error log, and cookie log (Buchner and Mulvenna, 1998). Typical web log record representing one web user access during one session is like the follows:

145.208.78.51-[26/Feb/2012:10:16:30-0500]"http://www.compusa.com/applications/ category/monitors/samsung.html HTTP/1.0 200 2781".

Table 1, shows the full interpretation of the previous web log record which contains information about one user's access to a web page during one session.

Field	Meaning	Example
Host/IP Address	Client IP address	145.208.78.51
User	User log name	'-' for anonymous user
Date and Time	Data, time, and time zone of user's request.	26/Feb/2012:10:16:30-0500
Requested URL	Uniform Resource Locator (URL) which has been requested by user.	http://www.compusa.com/applications/ category/monitors/samsung.html HTTP/1.0
Status	Status code returned to user	200 [series of success]
Bytes	Bytes transferred	2781 bytes

Table 1: AWeb Log File Record

Generally, usage data are collected by web servers in web data log files which are considered important sources for web traffic data. Web log registers user's navigational web pages during each session the user browses the web. Web usage mining researchers convert such usage logs into structured database tables through pre-processing to be able to apply web usage mining techniques and conclude the knowledge discovery. They represent each visited web page as an event and all the visited pages by a user during a period of time are collected as a transaction of sequence events for that user id. Thus, given a set of web pages P={a, b, c, d}, which represents a set of visited web pages by all users in a period of time, a web visit sequence for three users can be represented as the following three transactions in the format [transaction id, <sequence of web pages access>]: [T1, <adc>]; [T2, <abcd>]; [T3, bcda]. Mining such database may generate the frequent patter <ab> with support 70% which means that over 70% of users who visited page '*a*' also visited page '*b*'. The mining algorithms such as Apriori (Agrawal and Srikant, 1995) mines the frequent large pattern set L_i from the database table by generating the candidate items set C_i and select the most frequent items which meet the minimum support which defined by the end user as input for the Apriori algorithm, for each ith iteration the algorithm generates the next C_{i+1} by doing the join between L_i apriori-join L_i. Analyzing frequent patterns of users' visits to web pages can help business organisations to know customers' behaviors and their needs. Analyzing web log data and users' history registered data can also give valuable information to better organise and build web sites in appropriate way to target more customers. For example, if the web log shows that a web user's visits are most of the times to a certain type of pages such as http://....../......./electronics/laptops.html, which is a path for marketing laptops. This means that a particular user is interested in buying electronic devices especially laptops. Web usage mining also helps companies to get such information and develop their marketing policies to increase their sales.

1.1.3 Web Content Mining

Web content mining aims to mine useful knowledge from web page contents. Web contents can be in different formats: images, texts, audios, videos, hyperlinks, etc. Web content can also be unstructured (eg., text), semi-structured (eg., HTML), and structured (eg., XML, tables). Kosala et al. (2000) show that the majority of web contents are unstructured data contents. Unstructured web content can be represented as a set of words or texts. Each word can be Boolean or term frequency and can be reduced using different removal and selection techniques. Many of the proposed text mining techniques like information retrieval (Salton and Harman, 2003; Chowdhury, 2010), machine learning (Han and Kamber, 2000; Sebastiani, 2002), natural language processing (NLP) (Kao and Poteet, 2005), query answer question (Q/A) (Demner-Fushman, 2006) can be used to mine unstructured web content.

Multimedia web data mining includes analysis of large amounts of multimedia information which are located on the web to find special patterns or statistical relationships between multimedia data contents. Multimedia data represent any type of information that can be stored, and transmitted over the web in digital format like images, audios, videos, graphics, etc. Oh et al. (2003) propose a new technique to mine video data and extract interesting patterns from motions of mined videos. The mining algorithm can be applied on raw and surveillance videos. It works in two main steps; the first step the input data frames are dived into basic unites called segments in the structure of video. The second step the algorithm clusters the video segments into similar groups to conclude the knowledge and interesting patterns such as motions, object, and color. Mining multimedia web data is very important for many business companies to develop their marketing through identifying users' habits and observe their needs.

Semi-structured and structured web sources contain highly valuable rich information and include many of different types of data formats. Typical HTML page is considered semi-structured web data, which consists of text, image, hyperlink, structured data records, tables, and list. Such different types of data represent facts about the nature of web page contents worth to be extracted and mined for beneficial knowledge. Extracting structured data means obtaining regularly formatted data objects from the web, and creating structured database based on the extracted data. For example, extracting data object from Amazon website includes extracting each product which represents book and all the data attributes that are related to that book such as ISBN number, title, author, image, publishing date, etc.

1.2 Object-Oriented Data Warehouse (OODWH) Model

Kim (1990) defines the object-oriented database (OODB) as a logical structure of the real-world objects, constraints on them, and the relationships between the objects. In this thesis we propose to build object-oriented data warehouse (OODWH) model to store the extracted data context about different data objects such as products, lists, texts from different websites. Our advanced extraction system WebOMiner-3 extracts the object-oriented schema of each B2C website individually before building the integrated object-oriented data warehouse schema for capturing more comprehensive and detailed complexity of real world data, such as information related to different products browsed on B2C websites like versions, prices, images, or specifications. The main idea of object-oriented data warehouse is to provide a more natural way to represent data product items, providing a framework for manipulating the heterogeneous types of web contents and the complex relationships between them. The basic segment of object-oriented system is an object. Object is something existed, identified and distinguished, where each object consists of set attributes and unique identification identifies it than the other objects. The object can be a physical object such a computer, person, and book. The abstraction set of attributes, methods and operations which manipulate these attributes is called the class.

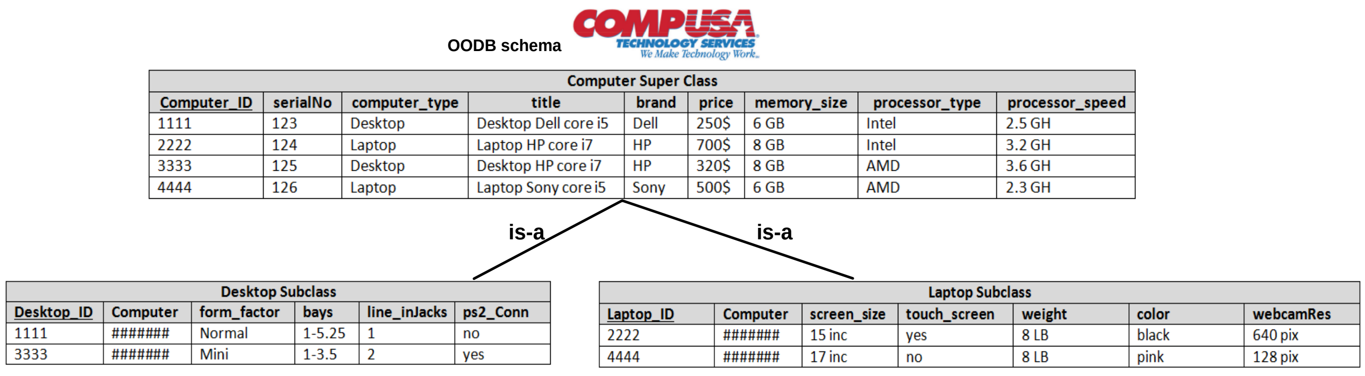
1.2.1 Definition of Object-Oriented Data Warehouse (OODWH) Model

Object-Oriented database represents a set of classes C_i which are connected together through the inheritance hierarchal relationships between the super-classes and sub-classes (Kemper and Moerkotte, 1994). Zhang (2011) defines object-oriented database model as the following, OODB is a set of hierarchal interconnected classes. Each class is defined as an ordered relation $\boldsymbol{C_i = (K, T, S, A, M, O)}$, where K is the instance (object) identifier, T is the class type, S is the super type of the class, A is the set of attributes, M is the set of methods, and O is the set of objects. Class inheritance hierarchy H represents the relationships between the classes.

In this thesis, we define a new object-oriented data warehouse (*OODWH*) model to integrate different OODB sources in one data warehouse that is suitable to the nature of the extracted data contents from different B2C websites. The data warehouse (DWH) is defined as subject-oriented, historical, non-volatile database integrating a number of database sources. The new *OODWH* model is defined as ***OODWH = (C, F, D),*** where C is set of hierarchal interconnected classes, where each class is defined as an ordered relation $\boldsymbol{C_i = (K, T, S, A, M, O)}$, where K is the instance (object) identifier, T is the class type, S is the super type of the class, A is the set of attributes, M is the set of methods, and O is the set of objects. **F** represents the fact tables, each fact table F_i is defined as ***Fi=(FK, SubA, SourceA, DateTimeA)***, where FK represents the set of foreign keys that are related directly to the primary keys of the dimension tables, SubA represents the subject attributes, SourceA represents the source of database from which the data were extracted, DateTimeA represents the date and time attributes which store the date and time of inserting data into the fact table. **D** represents the set of dimension tables, each dimension table D_i is defined as $\boldsymbol{D_i = (PK, DA)}$, where PK represents the primary key attributes of dimension table, and DA represents the other detailed attributes of dimension table.

Example OODWH Integration:

Let us suppose that we need to extract all the information that is related to all the *Laptops* and *Desktops* products from both the ‘compUSA’ and ‘BestBuy’ websites. WebOMiner-2 (Harunorrashid, 2012) is able to extract the OODWH schema of only one website and store the extracted data contents into relational database. Since the WebOMiner-2 extracts information about only one product item from only one source website, it would not able to integrate data of different products from different websites. Our new extraction system WebOMiner-3 generates the object-oriented database (OODB) schema for each given website individually, and then integrates different website’s schemas into one object-oriented data warehouse schema to store the data content from different websites. For example, let us suppose the OODB schema for the ‘compUSA’ website as shown in figure 2.



Computer Super Class

Computer_ID	serialNo	computer_type	title	brand	price	memory_size	processor_type	processor_speed
1111	123	Desktop	Desktop Dell core i5	Dell	250$	6 GB	Intel	2.5 GH
2222	124	Laptop	Laptop HP core i7	HP	700$	8 GB	Intel	3.2 GH
3333	125	Desktop	Desktop HP core i7	HP	320$	8 GB	AMD	3.6 GH
4444	126	Laptop	Laptop Sony core i5	Sony	500$	6 GB	AMD	2.3 GH

Desktop Subclass

Desktop_ID	Computer	form_factor	bays	line_inJacks	ps2_Conn
1111	#######	Normal	1-5.25	1	no
3333	#######	Mini	1-3.5	2	yes

Laptop Subclass

Laptop_ID	Computer	screen_size	touch_screen	weight	color	webcamRes
2222	#######	15 inc	yes	8 LB	black	640 pix
4444	#######	17 inc	no	8 LB	pink	128 pix

Figure 2: OODB schema of compUSA website.

As shown in figure 2 that the OODB schema for ‘compUSA’ website has three classes: C_1= Computer, C2= Desktop, and C3=Laptop and are defined as the following:

C1 (Computer) = (K, T, S, A, M, O)

K represents the instance (object) identifier such as Computer_ID=1111, Computer_ID=4444; **T** represents the class type which is ‘Computer’ ; **S** represents the super class which is ‘Root=null’ for computer class because the computer inherits the root class; **A** represents the common attributes between the 'Desktop' and 'Laptop' subclasses, so **A**= {Computer_ID, serialNo, computer_type, title, brand, price, memory_size, processor_type, processor_speed}; **M** represents the methods which manipulate the attributes such as setComputerID(String ID), getComputerID(),…, etc; **O** represents all the instances (objects) of type computer such as computer1, computer2, computer3, etc.

The second class represents the class of 'Laptop' and is defined as the following:

C2 (Laptop) = (K, T, S, A, M, O)

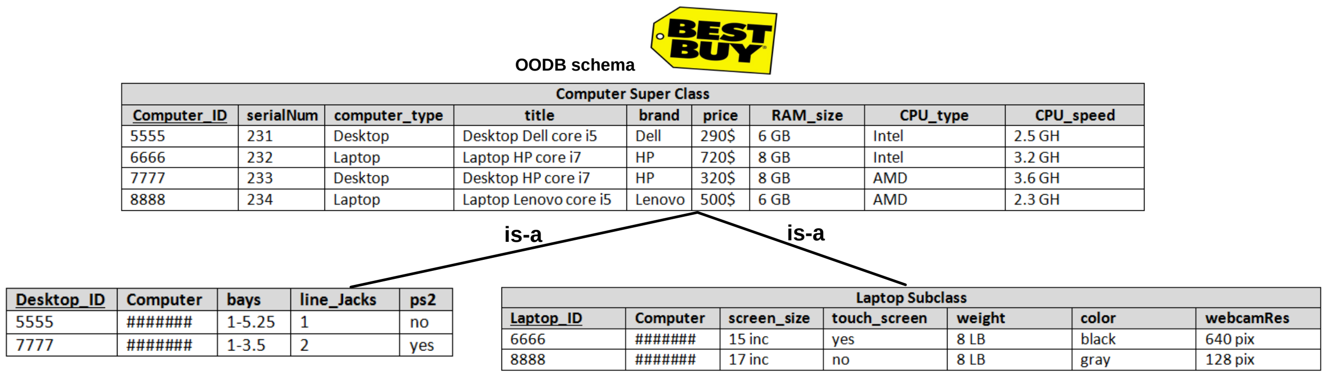
K represents the instance (object) identifier such as Laptop_ID=2222, Laptop_ID=4444; **T**: Laptop; **S** represents the super class of Laptop which is 'Computer' because the subclass 'Laptop' inherits the superclass 'Computer'; **A**: represents the distinct attributes of laptop which are not common with the desktop **A**= {Laptop_ID, Computer, screen_size, touch_screen, weight, color,webcamRes}; **M**: setScreenSize(String size), getScreenSize(), etc; **O**: laptop2, laptop4.

The third class represents the class of 'Desktop' and is defined as the following:

C3 (Desktop) = (K, T, S, A, M, O)

K represents the instance (object) identifier such as Desktop_ID=1111, Desktop_ID=3333, **T**: Desktop; **S** represents the super class of Desktop which is 'Computer' because the subclass 'Desktop' inherits the superclass 'Computer'; **A**: {Desktop_ID, Computer, form_factor, Bays, line_injack, PS2_Conn}; M: setFormFactor(String factor), getFormFactor(), etc; O: desktop1, desktop3.

Also the OODB schema of 'BestBuy' website has three classes C1(Computer), C2(Laptop), C3(Desktop) as shown in figure 3.



Computer Super Class

Computer_ID	serialNum	computer_type	title	brand	price	RAM_size	CPU_type	CPU_speed
5555	231	Desktop	Desktop Dell core i5	Dell	290$	6 GB	Intel	2.5 GH
6666	232	Laptop	Laptop HP core i7	HP	720$	8 GB	Intel	3.2 GH
7777	233	Desktop	Desktop HP core i7	HP	320$	8 GB	AMD	3.6 GH
8888	234	Laptop	Laptop Lenovo core i5	Lenovo	500$	6 GB	AMD	2.3 GH

Desktop_ID	Computer	bays	line_Jacks	ps2
5555	#######	1-5.25	1	no
7777	#######	1-3.5	2	yes

Laptop Subclass

Laptop_ID	Computer	screen_size	touch_screen	weight	color	webcamRes
6666	#######	15 inc	yes	8 LB	black	640 pix
8888	#######	17 inc	no	8 LB	gray	128 pix

Figure 3: OODB schema of BestBuy website.

The WebOMiner-3 integrates these two OODB schemas into one OODWH schema by matching the attributes of the two sources schemas as shown in figure 4. The OODWH is defined as following:

OODWH=(C, F, D)

Where **C** represents the three classes which are C1=Computer_dim, C2=Desktop_dim, C3=Laptop_Dim; and each Ci is defined as:

Ci = (K, T, S, A, M, O)

The superclass C_1 Computer contains the common attributes between the 'Desktop' and 'Laptop' subclasses in the two websites sources and is defined as ***C1(Computer_dim)= (K, T, S, A, M, O)***,

K represents the instance (object) identifier such as ComputerID=2222; **T** represents the class type which is 'Computer_dim' ; **S** represents the super class which is 'Root=null' for computer superclass; **A**={ ComputerID, title, memorySize, processorType, processorSpeed};

M represents the methods such as setTitle(String t), getTitle();**O** represents all the instances (objects) of type computer such as computer1, computer2, computer3, etc.

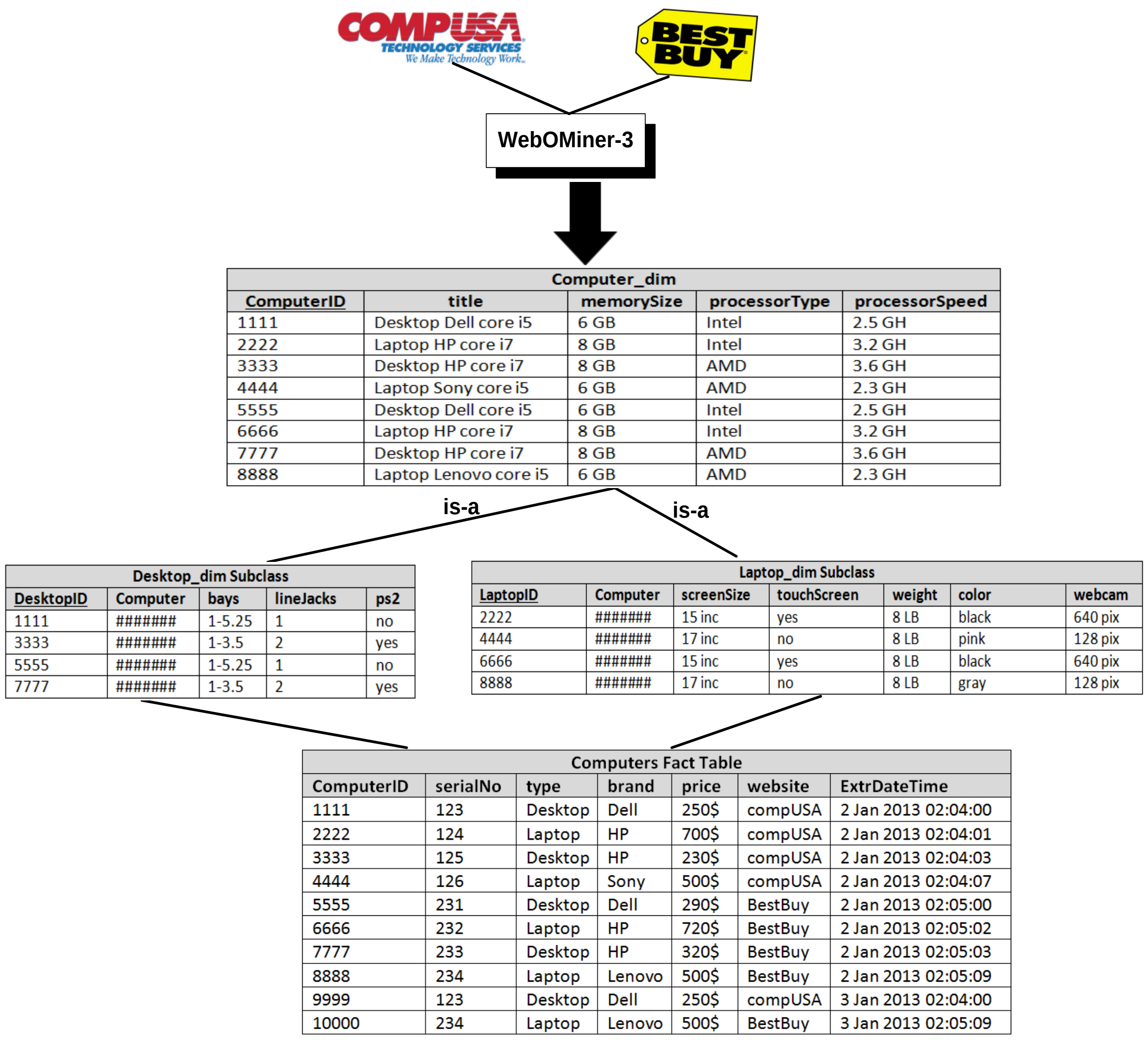
The C_2 Laptop contains the distinct attributes of 'Laptop' subclass table and is defined as ***C1(Laptop_dim) = (K, T, S, A, M, O)***, **K** represents the instance (object) identifier such as LaptopID=8888; **T** represents the class type which is 'Laptop' ; **S** represents the super class of Laptop which is 'Computer_dim' because the subclass 'Laptop_dim' inherits the superclass 'Computer_dim';

A: represents the distinct attributes of laptop which are not common with the desktop from both CompUSA and BestBuy websites **A**={LaptopID, Computer, screenSize, touchScreen, weight, color, webCam}; **O** represents all the instances (objects) of type laptop such as laptop2, laptop8.

The C_3 Desktop contains the distinct attributes of 'Desktop_dim' subclass table and is defined as ***C1(Desktop_dim)= (K, T, S, A, M, O)***, **K** represents the instance (object) identifier such as DesktopID=7777; **T**: represents type of subclass which is 'Desktop'; **S** represents the super class of Desktop which is 'Computer_dim' because the subclass 'Desktop_dim' inherits the superclass 'Computer_dim'; **A**: represents the distinct attributes of desktop which are not common with the laptop from both CompUSA and BestBuy websites **A**= {DesktopID, Computer, bays, linJack, ps2}; **O** represents all the instances (objects) of type desktop such as desktop1, desktop7.

F represents the fact tables which are {ComputersFact, TuplesFact}; each fact table Fi is represented as ***Fi= (FK, subA, sourceA, DateTimeA}***, **FK** represents the foreign keys = {ComputerID, serialNo}; **subA** represents the subject attributes = {price, type, brand}; **sourceA** represents the source website from which the data were extracted = {compUSA, BestBuy}; **DateTimeA** represents the date and time of the extraction process.

D represents dimension tables, each dimension table Di is represented as ***Di= (PK, DA)***, **PK** represents the primary key attributes for example in the Laptop_Dim table the primary key is LaptopID; **DA** represents the other detailed attributes which describe the tuple inside the table, in the List_Dim table the **DA**= {link, title}. The complete D in OODWH in this example is { Computer_dim, Desktop_dim, Laptop_Dim, List_Dim, Text_Dim, Form_Fact, Noise_Fact}. Figure 5 shows the complete OODWH schema.



Computer_dim

ComputerID	title	memorySize	processorType	processorSpeed
1111	Desktop Dell core i5	6 GB	Intel	2.5 GH
2222	Laptop HP core i7	8 GB	Intel	3.2 GH
3333	Desktop HP core i7	8 GB	AMD	3.6 GH
4444	Laptop Sony core i5	6 GB	AMD	2.3 GH
5555	Desktop Dell core i5	6 GB	Intel	2.5 GH
6666	Laptop HP core i7	8 GB	Intel	3.2 GH
7777	Desktop HP core i7	8 GB	AMD	3.6 GH
8888	Laptop Lenovo core i5	6 GB	AMD	2.3 GH

Desktop_dim Subclass

DesktopID	Computer	bays	lineJacks	ps2
1111	#######	1-5.25	1	no
3333	#######	1-3.5	2	yes
5555	#######	1-5.25	1	no
7777	#######	1-3.5	2	yes

Laptop_dim Subclass

LaptopID	Computer	screenSize	touchScreen	weight	color	webcam
2222	#######	15 inc	yes	8 LB	black	640 pix
4444	#######	17 inc	no	8 LB	pink	128 pix
6666	#######	15 inc	yes	8 LB	black	640 pix
8888	#######	17 inc	no	8 LB	gray	128 pix

Computers Fact Table

ComputerID	serialNo	type	brand	price	website	ExtrDateTime
1111	123	Desktop	Dell	250$	compUSA	2 Jan 2013 02:04:00
2222	124	Laptop	HP	700$	compUSA	2 Jan 2013 02:04:01
3333	125	Desktop	HP	230$	compUSA	2 Jan 2013 02:04:03
4444	126	Laptop	Sony	500$	compUSA	2 Jan 2013 02:04:07
5555	231	Desktop	Dell	290$	BestBuy	2 Jan 2013 02:05:00
6666	232	Laptop	HP	720$	BestBuy	2 Jan 2013 02:05:02
7777	233	Desktop	HP	320$	BestBuy	2 Jan 2013 02:05:03
8888	234	Laptop	Lenovo	500$	BestBuy	2 Jan 2013 02:05:09
9999	123	Desktop	Dell	250$	compUSA	3 Jan 2013 02:04:00
10000	234	Laptop	Lenovo	500$	BestBuy	3 Jan 2013 02:05:09

Figure 4: OODWH Integration of Computer product from two websites.

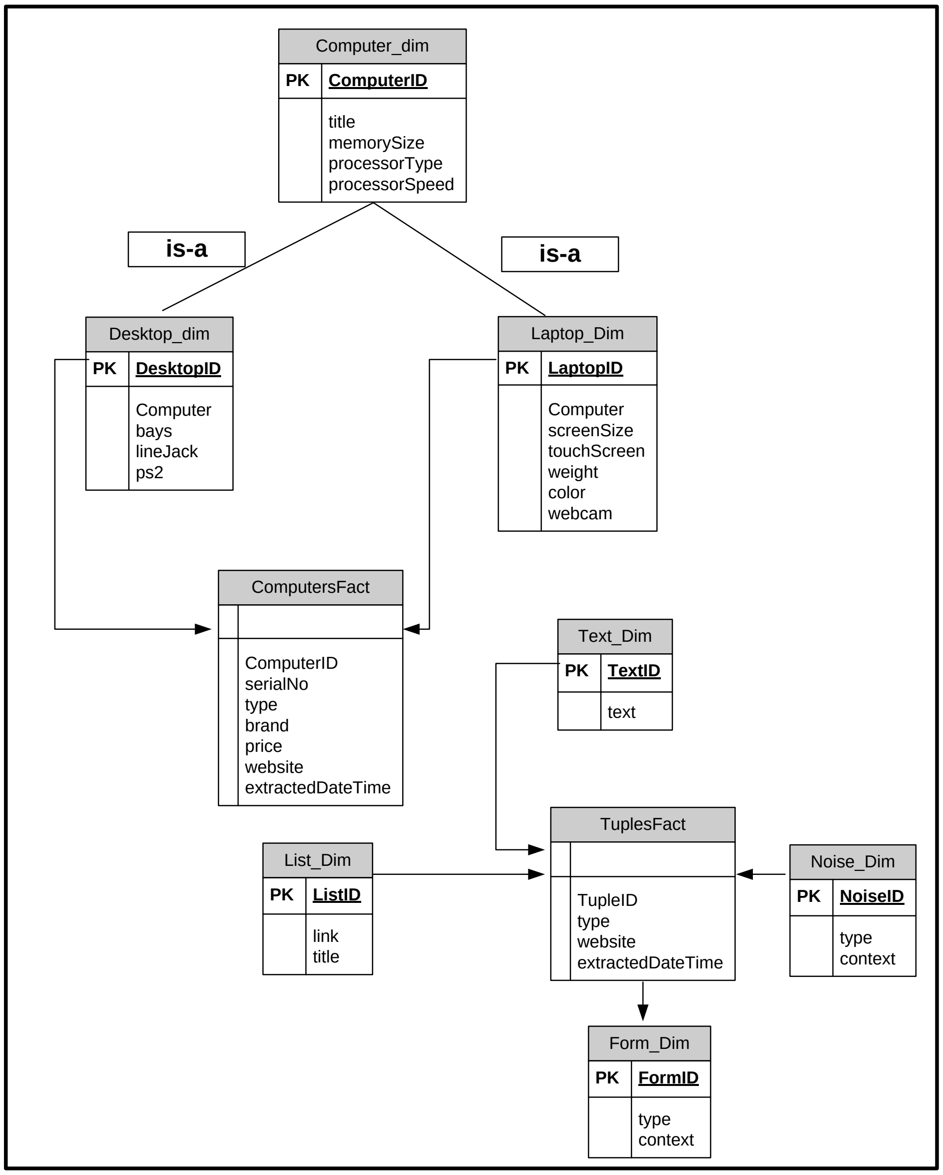


Figure 5: OODWH schema for integrating two websites schemas.

1.2.2 Aspects of Object-Oriented Model (OOM)

The main aspects of any object-oriented model include define the classes and creation of the objects which involve actions of the object’s attributes. In addition, to the hierarchal relationships between these objects, inheritance, association, aggregation, and encapsulation.

▪ **Inheritance Relationships**

Inheritance relationship is a special relationship in which the definition of a class is based on the definition of another existing class. For example, if one class inherits another class then the first class is called the *subclass* and the second class is called the *superclass*. A subclass contains all the attributes and methods of superclass, and has its own attributes and methods. This means, every subclass can be a superclass but the opposite is not always true. Figure 6 shows the inheritance relationship between the 'Computer' superclass and 'Desktop' and 'Laptop' subclasses. As shown in the figure 6, there is an inheritance relationship is called 'is-a' relationship and can be interpreted as the following: '*Every laptop or desktop is-a computer, but not every computer can be laptop or desktop*'

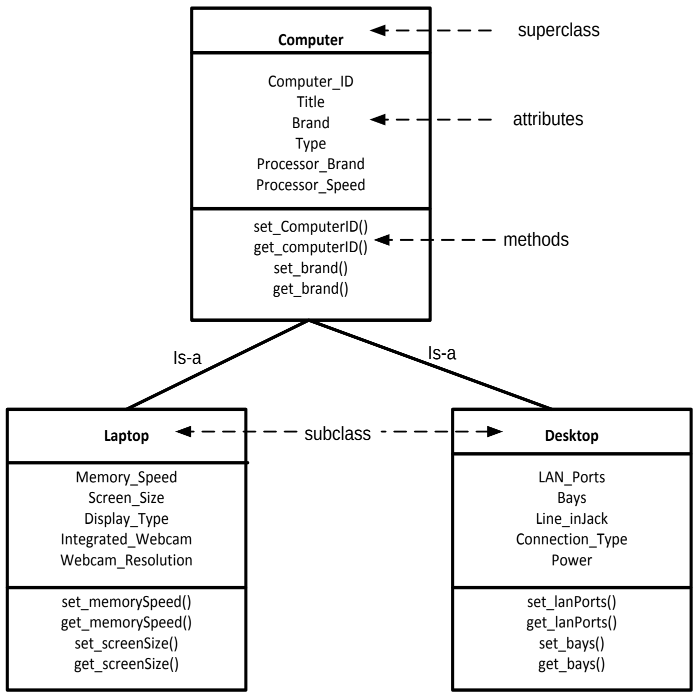


Figure 6: Inheritance relationships between computer products.

- **Association Relationships**

Association refers to the connection relationship between classes. The classes are connected through links and each link has cardinality such as, one-to-one, one-to-many, and many-to-many for example there is one-to-one relationship between the ‘Laptop’ class and ‘Computer’, where the laptop can be only one computer, and the computer can be only one laptop. In the ‘Student’ and ‘Course’ classes relationship, there is association relationship with a cardinality of type many-to-many, where the student can register for more than one course, and the course can be taken by more than one student.

- **Aggregation Hierarchies**

Aggregation is special type of association relationship and called ‘part-of’ relationship in which the whole object consists of small parts (objects). For example, the aggregation relationship between the PC (Personal Computer) as a whole and its components which defined as a separate classes such as hard disk, CPU, monitor, keyboard as shown in figure 7.

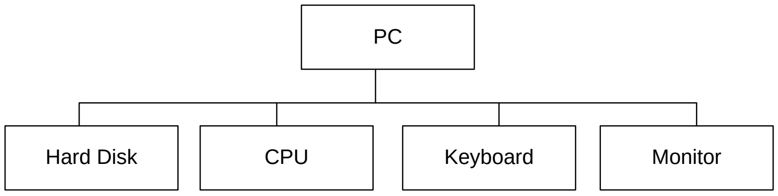


Figure 7: Aggregation relationship between PC and its parts.

- **Encapsulation**

The concept of encapsulation in object-oriented programing language or database refers to the idea of encapsulate all the attributes, operations and methods which are related to a specific object through hiding their implementation, and define a small interface to deal with the object’s attributes. For example, the laptop product object contains a set of attributes such as brand, processor type, memory size, and methods which manipulate the attributes such as setBrand(String B) method which assigns the brand value for the brand attribute. The object encapsulates all its attributes, methods, and operations in one class.

1.2.3 Components of Object-Oriented Database Model

The object-oriented database model consists of the following main concepts:

1) Class:

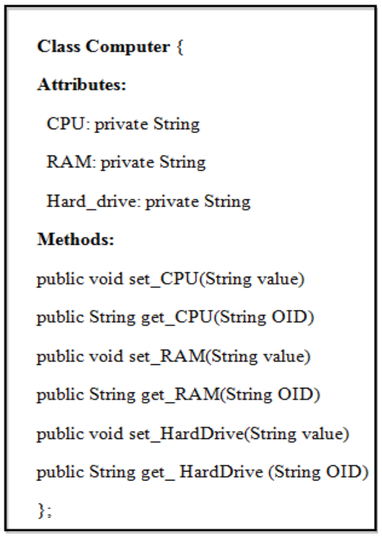
Is a general abstraction representation of all the instantiated objects which share the same attributes, methods and operations. For example, all computers desktops, laptops, and pads share the following attributes: CPU, RAM, Hard Drive. The class defines all the general common attributes and methods of the objects which are applicable to be inherited. Figure 8 shows the class definition of computer object.

2) Attributes:

The data items which describe the properties and specifications of a specific object. For example, *Computer Type, Computer Brand, CPU Speed, RAM Capacity, Hard Drive Capacity* are attributes *of* any computer object.

3) Methods:

The procedures which assign and retrieve the attribute values. For example, the procedure *setCPU_Type( )* assigns the value of CPU type attribute.



```
Class Computer {
Attributes:
  CPU: private String
  RAM: private String
  Hard_drive: private String
Methods:
public void set_CPU(String value)
public String get_CPU(String OID)
public void set_RAM(String value)
public String get_RAM(String OID)
public void set_HardDrive(String value)
public String get_ HardDrive (String OID)
};
```

Figure 8: Class Definition of Computer Objet.

4) The object and object identifier:

Object is a real world entity as ‘Computer’ and represents an instance of the abstraction definition of the class. In object-oriented database every object has a unique identifier called OID to distinguish each object individually, and this identifier remains unique through the whole life time of the object.

5) Class Inheritance Hierarchy Relationships:

Class hierarchy represents the classification of objects class type. Data objects with similar properties and specifications are grouped together and described by class type. Similar objects mean objects have similar attributes, methods, features, and behaviors. If objects share in common similar attributes but also slightly differ in other attributes, they should belong to different class type. The relationship between the classes of slightly different objects is called the hierarchal relationship. Class inheritance represents the relationship between the classes in the object database, for example the child (subclass) inherits the parent (superclass). Figure 9 shows the hierarchal object-oriented relationship of B2C websites.

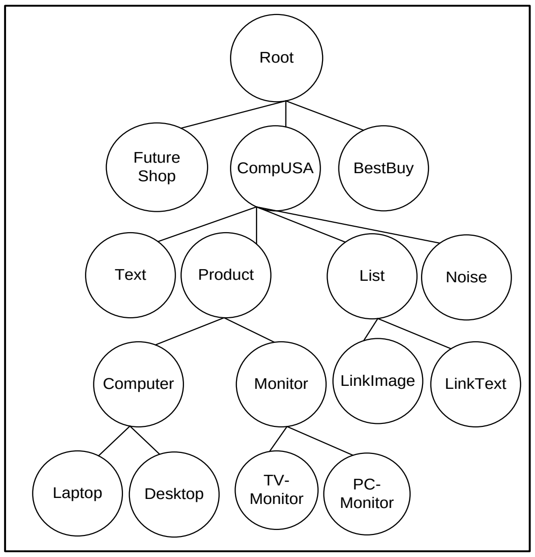


Figure 9: Inheritance hierarchy relationship of different data objects.

1.3 Types of Extracted Web Pages

Liu (2006) classifies data rich web pages in e-commerce into two types: list pages and detailed pages. Usually such pages containing rich data come from underlying structured database and embedded templates hidden beneath HTML tags. In this thesis, we are mainly interested in list and detailed product web pages.

1.3.1 List Pages

List page is a page that contains a list of continuous similar data objects such as product object, list object, text object. Figure 10 shows an example of list product page from e-commerce web site for electronic devices (CompUSA.com). As figure 10 shows, the page contains two main data regions, data region1 and data region 2, which are labelled by continuous line. The *data region* is defined as a group of adjacent and similar data blocks which share the same parent tag node within a particular area of the page. *Data block* is defined as a sequence of adjacent HTML tag nodes which share the same parent tag node and all together are related to a distinct data item inside the page such as monitor product object. Figure 10 shows a data product object for SAMSUNG laptop computer with a price of 449.99$ labelled by dashed line. Each data block represents a data record (tuple). For example, in figure 10 data region1 contains a group of sibling data blocks for laptops products objects. Each laptop object is presented by adjacent HTML tags and attributes such as <image><price><tittle><product number> <brand> which are related together.

1.3.2 Detailed Pages

Detailed page is a page that contains details of information about a single data object. Figure 11 shows an example of detailed page which gives detailed information about laptop product. As shown in figure 11, the page contains information about the description of the product, image, title, price, specifications in different locations of detailed web page.

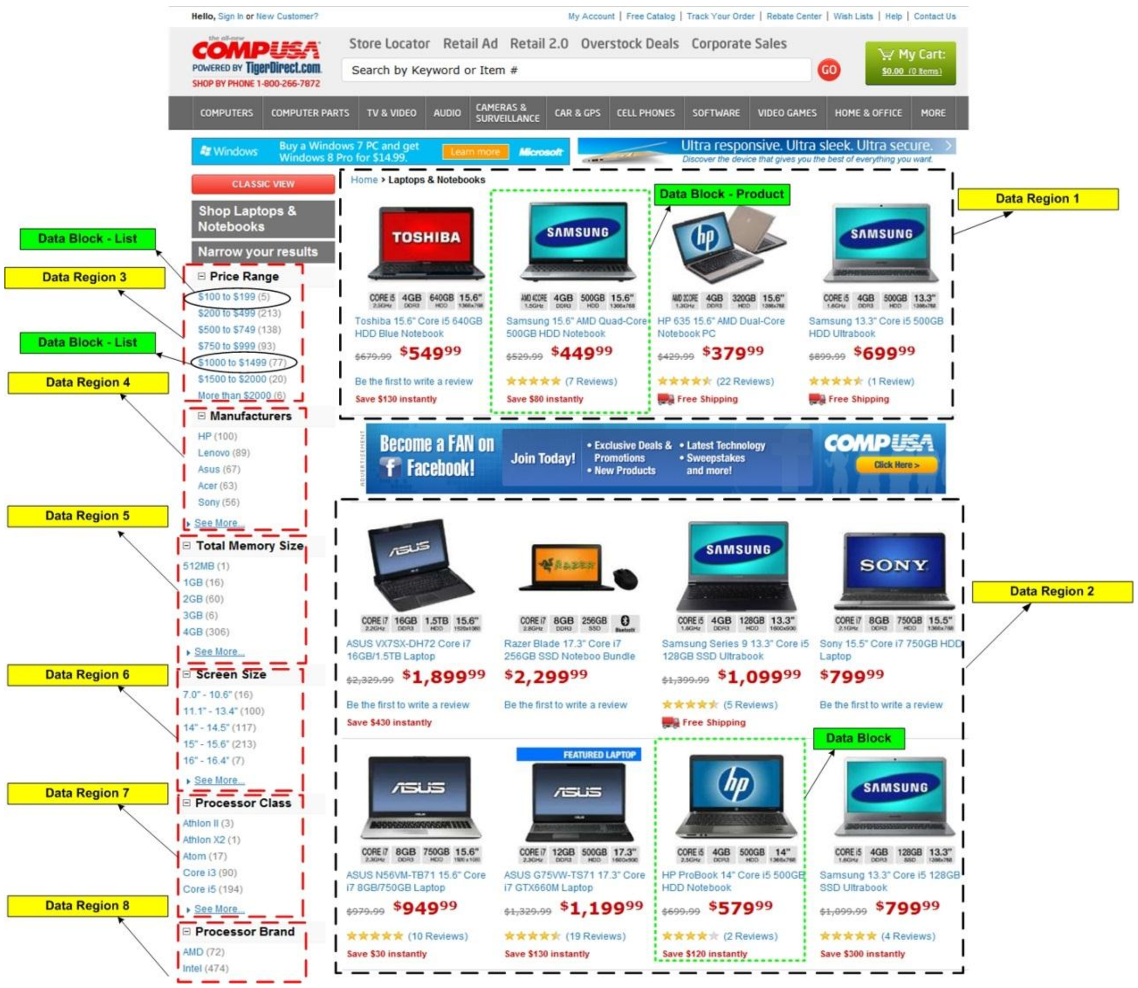
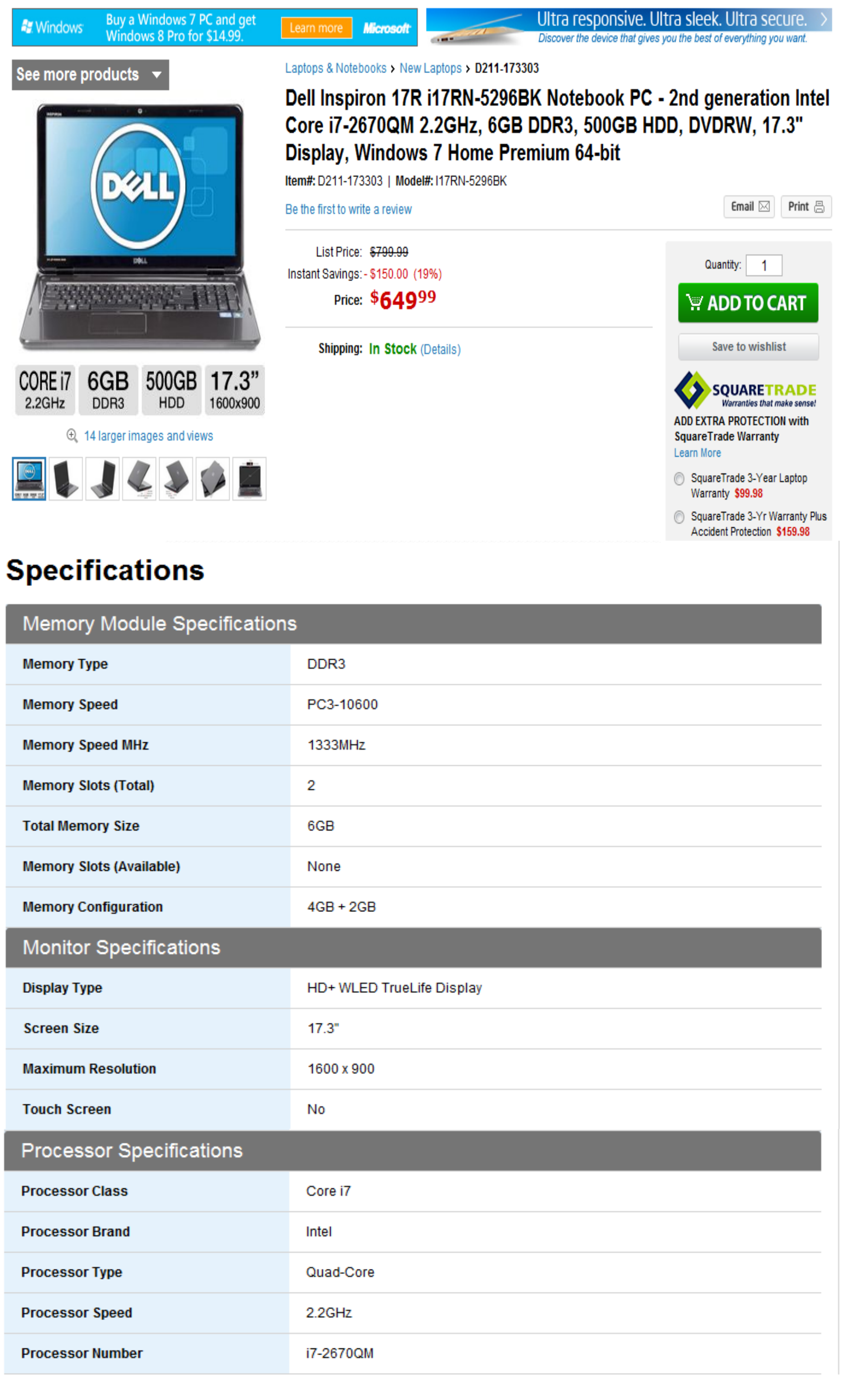


Figure 10: List product web page.



Windows Buy a Windows 7 PC and get Windows 8 Pro for $14.99. Learn more Microsoft

Ultra responsive. Ultra sleek. Ultra secure. Discover the device that gives you the best of everything you want.

See more products

Laptops & Notebooks › New Laptops › D211-173303

Dell Inspiron 17R i17RN-5296BK Notebook PC - 2nd generation Intel Core i7-2670QM 2.2GHz, 6GB DDR3, 500GB HDD, DVDRW, 17.3" Display, Windows 7 Home Premium 64-bit

Item#: D211-173303 | Model#: I17RN-5296BK

Be the first to write a review

Email Print

CORE i7 2.2GHz | 6GB DDR3 | 500GB HDD | 17.3" 1600x900

14 larger images and views

List Price: ~~$799.99~~
Instant Savings: - $150.00 (19%)
Price: $649.99

Shipping: In Stock (Details)

Quantity: 1

ADD TO CART

Save to wishlist

SQUARETRADE Warranties that make sense!
ADD EXTRA PROTECTION with SquareTrade Warranty
Learn More
SquareTrade 3-Year Laptop Warranty $99.98
SquareTrade 3-Yr Warranty Plus Accident Protection $159.98

Specifications

Memory Module Specifications	
Memory Type	DDR3
Memory Speed	PC3-10600
Memory Speed MHz	1333MHz
Memory Slots (Total)	2
Total Memory Size	6GB
Memory Slots (Available)	None
Memory Configuration	4GB + 2GB
Monitor Specifications	
Display Type	HD+ WLED TrueLife Display
Screen Size	17.3"
Maximum Resolution	1600 x 900
Touch Screen	No
Processor Specifications	
Processor Class	Core i7
Processor Brand	Intel
Processor Type	Quad-Core
Processor Speed	2.2GHz
Processor Number	i7-2670QM

Figure 11: Detailed product web page

Example Queries that can be answered by WebOMiner-3:

Develop a comparative shopping system that is able to answer the following types of queries:

1. List all Dell core i5 and 6GB RAM laptops prices which are offered now by ‘CompUSA’, ‘BestBuy’, ‘Homedepto’, ‘Shopxscargo’, and ‘Factorydirect’ websites, and compare the prices with the previous month prices for the same laptop specifications. The WebOMiner and WebOMiner-2 systems cannot answer such query because it requires an integrated and historical data warehouse.
2. Make a comparison of HP laptops prices which have been offered by ‘CompUSA’, ‘BestBuy’, ‘Homedepto’, ‘Shopxscargo’, and ‘Factorydirect’ websites since the last two years until today grouped by the months of the year. This query requires an integrated and historical data warehouse.
3. List all cpu type, cpu speed, model brand, model title, features of Desktops have been offered by the previous five websites ordered by year of manufactured and cpu speed. This query requires an integrated and historical data warehouse.
4. What is the best place right now to go and buy Dell Laptop with Intel CORE i7 and 8 GB Ram among the ‘CompUSA’, ‘BestBuy’, ‘Factorydirect’ websites?. The WebOMiner and WebOMiner-2 systems cannot answer such query because it requires an integrated data warehouse.
5. Is there any sale on Sony core i7 8 GB RAM in ‘Homedepto’ website compare to its’ prices of the previous year at the same current month?. The WebOMiner and WebOMiner-2 systems cannot answer such query because it requires a historical data warehouse.

To answer such queries directly from the web is a difficult task to do, where the user should browse each web site and write down the price of the product which the user is looking for, and then do the manual comparison to decide which web site offers the cheapest price of the required product. For this reason, we propose to advance the WebOMiner system to better automate the extraction process using DOM Tree tag nodes matching technique, and then identify each extracted data block type through matching with the right NFA structure, and finally store the identified extracted data tuple in the right place in the proposed historical object-oriented data warehouse (OODW) repository. Then post such previous queries on the OODW to get the answers.

We need to distinguish our proposed work from similar related works by “Web Query Interface” (Bornhovd and Buchmann, 1999; Liu, 2006) and “Web Service” (Walchhofer et al., 2010). Web query interface provides user a global query interface to query data from multiple data sources without physically creating database or data warehouse. Such interfaces extract queries’ results from web or any other sources on the fly and return the results directly to the end user without intermediate database storage. The main problem with query interface is it needs huge amount of costly efforts to filter the data from different data sources and to guarantee the consistency between them in order to return results for the posted query. In addition, in many cases query interface retrieves inaccurate results. So it is clear that query interface cannot answer our target queries. Web service is a new technique used in ‘Semantic Web’ to extract and integrate data from business websites. Web service does not deal with free HTML web information, besides it is a service and needs to be bought from service provider. For example, ‘yahoo tool bar’ is a web service offered by ‘Yahoo’ company to be used by other websites and users for search options. Web service does not hold historical data, so it cannot answer our target queries. Our proposed system deals with free HTML pages, and stores historical data and it is promising to answer our target queries.

1.4 Thesis Problem Statement

A large number of list product web pages such as ‘BestBuy’ website pages contain essential information in structured format. Such structured information is called data block or tuple and describe a specific item on the web. Such similar data blocks are adjacent and located in one data region in list product web page. Each data block has a detailed web page that describes the product in details and lists all its specifications. For example, once the customer logs in to *CompUSA* website and clicks on the web list computer laptops web page will find many of laptop blocks, each laptop block is described by summarised information called attributes such as the type of the laptop which is represented by title, brand, manufactured company, serial number, price, image. Also each laptop block has a link to the detail web page which describes the properties and specifications of laptop.

The problem statement of mining data objects can be summarised as follows*:* Let us suppose that we have a set of product list web pages $P= \{p_1, p_2,...., p_n\}$ from different B2C web sites. p_i contains a set of different types of data blocks $B=\{db_1, db_2,..., db_n\}$. Each data block db_i is embedded underneath a set of HTML tags, where some data blocks $\{db_1, db_2....db_l\}$ have the same HTML tags template, and some of them have different HTML template. Each data block db_i consists of one or more data attribute f_{ij}. Given a set of list and detailed product web pages P (Desktop and Laptop list web pages) from different websites W, the WebOMiner-3 generates the object-oriented database (OODB) schema for each website w individually which combines the information about all the products (Desktops and Laptops) items in the website. Then the WebOMiner-3 integrates the generated OODB schemas of websites by matching the common attributes between the product items and put the them in one superclass, and build a subclass for each product item which contains the distinct attributes that are not common with any other product such as ‘Laptop’ and ‘Desktop’ subclasses, then builds the fact table which stores the historical information about the products items from different websites based on the main subject attributes such as price, brand, type attributes. For a product (Laptop or Desktop) list web page p_i given as input, the new advanced WebOMiner-3 system would crawl p_i from the web and store it in local machine, and then it cleans the HTML code of p_i to build the DOM tree of the given web page. WebOMiner-3 traverses the generated DOM tree to automatically extract and mine each data object ob_i in each p_i based on the similarity of HTML tag nodes of those similar data blocks that share the same root tag node, and then automatically identify type of each data block (tuple) db_i and store it in the appropriate place in historical object-oriented data warehouse repository.

Mining such data blocks and their attributes from the web and storing them in data warehouse repository is very important and beneficial for customers to develop the web market companies. In addition locating such information about data items in one data repository has an important role to provide a comparative shopping service for customers allowing them to do a comparison between the products that they intent to buy. Also such information helps to develop many of the web services and develop online websites and e-markets which have good impact on both the merchants and the customers. Actually, extracting such structured data tuples from the web contents is a real challenge and not an easy task due to complex structure of the web pages that contain these objects, beside that there are many of different types and formats of web data contents which represent the data objects that are targeted to be extracted.

1.5 Thesis Contributions

This thesis includes many of pre-processing steps to automatically extract web contents that are not addressed by Annoni and Ezeife (2009); Mutsuddy (2010); Harunorrashid (2012); Ezeife and Mutsuddy (2013). We propose advancing the WebOMiner system for extracting and mining of web contents. The new modifications and enhancements include adding a new extraction procedure based on the similarity of DOM tree tag nodes' patterns to identify data blocks and their data regions inside the list product web page, building a full automatic Non Deterministic Finite Automate (NFA) structure for each type of similar data block based on the generated regular expression of frequent object. The new modifications also include proposing a new historical object-oriented data warehouse to integrate and store the extracted data tuples from different web pages in the domain of B2C websites. The following are main contributions of this thesis:

1. Propose a new extraction technique based on the similarity of the DOM tree tag nodes string patterns of the targeted data blocks inside a list product web page. The new technique compares the HTML pattern for each data block with other blocks' patterns and extracts the blocks that have the same HTML pattern and are neighbours. The WebOMiner and WebOMiner-2 do not have a clear definition for the block level and non-block level data blocks which are used to extract data records from web page; in addition, their definition is not applicable for some web pages like BestBuy website. Also the WebOMiner and WebOMiner-2 do not specify the boundary of data block whether level or non-level data block. For example, they do not provide any algorithm to specify when the extraction process starts and when or where it ends.
2. Build a fully automatic Non Deterministic Finite Automata (NFA) structure for each type of the extracted data block such as product, list, text, form, noise. The new concept of fully automatic NFA will be accomplished through frequent object mining (FOM) technique. The WebOMiner-2 depends only on the frequency of data attributes and does not take into the consideration the sequence order of the attributes. For this reason, the WebOMiner-2 generates many of extra regular expressions which need to be built as NFA structures because they are candidate to be data records. For example WebOMiner-2 considers the pattern 'image title' is different than the pattern 'title image' and generates two different regular expressions for them and two NFA structures. The WebOMiner-3 solves the sequence order of the attributes problem by preserving the frequency and sequence order of the attributes through build the frequent object tree (FO-Tree) which gives the unique paths for each candidate data record.

3. Modify data block and data region definitions which have been used by (Mutsuddy, 2010; Ezeife and Mutsuddy, 2013). To find the block level and non-block level data blocks inside the page, the WebOMiner and WebOMiner-2 need to scan every tag node in the DOM-Tree of the web page which is a very costly process. Also the proposed definitions are not applicable for some web pages like ‘BestBuy’ website. For that reasons, we modify the definitions of both data block and data region to save some comparisons during the extraction process. The new definitions of data block and data region are applicable for all the list and detailed product web page. Besides that the definitions are more efficient during the identification of data blocks to be extracted from DOM-Tree.
4. We combine between HTML tag names and their attributes to guarantee building a unique structure for each type of tuples like product tuple, list tuple, text tuple. For example, we benefit from the HTML tag such as <a> tag and its attribute ‘title= Dell laptop core i7’ to know that the current processed data block is a laptop product item with the title= ‘Dell laptop core i7’ and has a link refers to the value which is stored in the src attribute ‘src=compusa\dell1.asp’.
5. We identify noise data block and prevent inserting them in data warehouse.
6. Extract the specifications for each data block from the detailed product web page.
7. Build automatic object-oriented data warehouse to store the historical information about data tuples, integrate data tuples from different web pages, and use this data warehouse for further mining processes and manipulations.
8. Build a GUI interface allows the end user to post a query to compare product items from different web pages and show the differences between them.

1.6 Outline of the Thesis

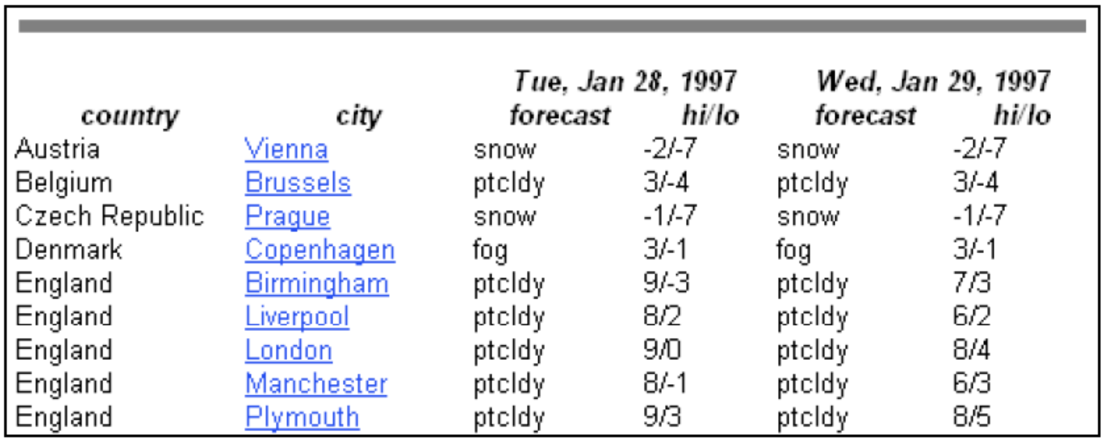
The reminder of the thesis is organised as follows: chapter 2 reviews related work to this thesis proposal. Chapter 3 includes details discussion of the new advanced WebOMiner-3 system along with all the proposed algorithms. Chapter 4 discusses performance analysis and experimental results. Chapter 5 draws the conclusion of this research and discusses future work.

CHAPTER 2- Related Work

Our research area in web content mining is related to Information Extraction (IE), which focuses on automatic structured data extraction techniques. There are three main categories of structured web contents extraction techniques: manual wrapper generation, semi-automatic or supervised learning wrapper, automatic wrapper generation. Related works in this chapter are organised as manual wrapper generation in section 2.1, supervised wrapper induction in section 2.2, and automatic wrapper generation in section 2.3.

2.1 Manual Wrapper Generation

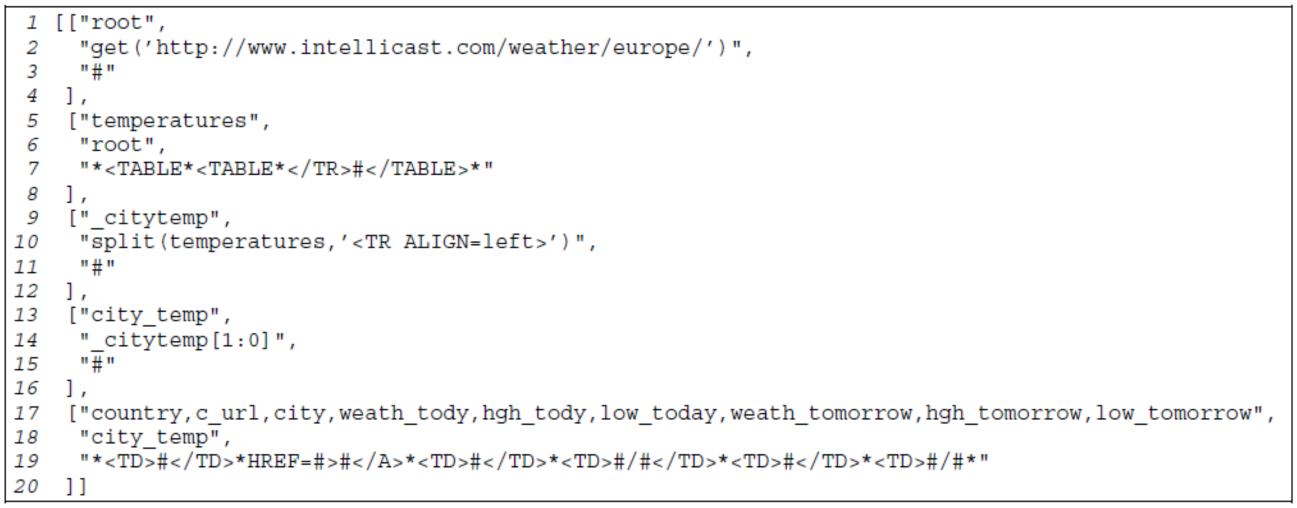
Wrapper is specialized routine program to extract data from web pages and convert the information into structured format like relational database, XML. Manual wrapper is built by user or programmer developer who writes the code of the extraction rules, and then applies these rules to extract the data of interest from web pages. Many of research works have been done on generating manual wrappers such as (Hammer et al., 1997), Xwrap (Liu et al., 2000), W4F (Sahuguet and Azavant, 1999). Hammer et al., (1997) propose a manual wrapper program to extract data from HTML pages as objects and store them in structured database. The proposed wrapper takes as input a descriptor that specifies types of data of interest, and how they should be identified and wrapped into data objects. The following example illustrates the proposed system by (Hammer et al., 1997). Let us suppose that there is a web application provides information about the weather status in different cities of the world as shown in figure 12.



		Tue, Jan 28, 1997		Wed, Jan 29, 1997	
country	city	forecast	hi/lo	forecast	hi/lo
Austria	Vienna	snow	-2/-7	snow	-2/-7
Belgium	Brussels	ptcldy	3/-4	ptcldy	3/-4
Czech Republic	Prague	snow	-1/-7	snow	-1/-7
Denmark	Copenhagen	fog	3/-1	fog	3/-1
England	Birmingham	ptcldy	9/-3	ptcldy	7/3
England	Liverpool	ptcldy	8/2	ptcldy	6/2
England	London	ptcldy	9/0	ptcldy	8/4
England	Manchester	ptcldy	8/-1	ptcldy	6/3
England	Plymouth	ptcldy	9/3	ptcldy	8/5

Figure 12: A snapshot of weather application source (Hammer et al., 1997).

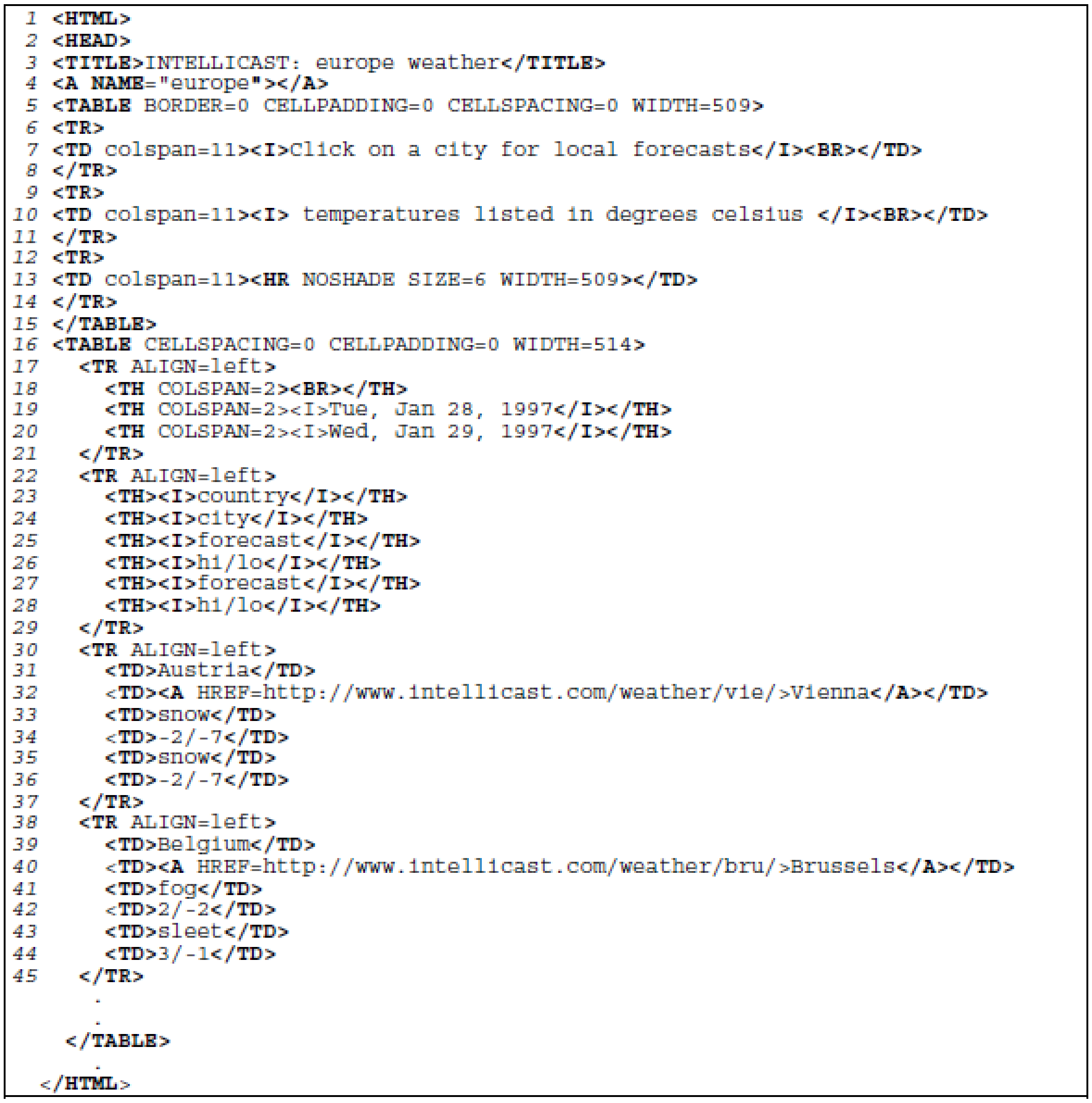
Since this web application always retrieves the web page as a search result, it cannot directly answer the queries such as (what is the temperature for Liverpool for Jan 29, 1997?). For this reason, we need to extract the contents of this web application, and store them in structured database to be able to answer such queries. The proposed manual extraction system parses the HTML code of this web page based on the extraction specification file shown in figure 13.



```
[["root",
  "get('http://www.intellicast.com/weather/europe/')",
  "#"
 ],
 ["temperatures",
  "root",
  "*<TABLE*<TABLE*</TR>#</TABLE>*"
 ],
 ["_citytemp",
  "split(temperatures,'<TR ALIGN=left>')",
  "#"
 ],
 ["city_temp",
  "_citytemp[1:0]",
  "#"
 ],
 ["country,c_url,city,weath_tody,hgh_tody,low_today,weath_tomorrow,hgh_tomorrow,low_tomorrow",
  "city_temp",
  "*<TD>#</TD>*HREF=#>#</A>*<TD>#</TD>*<TD>#/#</TD>*<TD>#</TD>*<TD>#/#*"
 ]]
```

Figure 13: A sample of extraction specification file (Hammer et al., 1997).

The specification file contains a sequence of commands; each command represents the current extraction step. The command is of the format [variables, source, pattern]. Where the source represents the input HTML code that should be parsed, pattern specifies the text of interest from the source, and variables store the extracted results. The extraction process begins by the initial command (lines 1- 4 figure 13) by fetching the contents of the URL given in line 2 and store them in the variable root as shown in figure 13. Line 3 (figure 13) '#' means extract all the contents of the given URL. The second command (lines 5-8) applies the pattern in line 7 to the source root and stores the result in variable called *temperatures*. The pattern at line 7 means discard everything until the first occurrence of </TR> tag after the two consecutive tags <TABLE> <TABLE>, and then starts the extraction until the beginning of the tag </TABLE> (i.e. extract the data between the command </TR># and </TABLE>). Now the variable *temperatures* contains the information stored in line 22 until line 45 (figure 14). The third command (lines 9- 12 figure 13) splits the contents of the *temperatures* variable into 'sets' of text using the split string <TR ALIGN=left>. The result of sets will be stored in variable *_citytemp*. The command 4 (lines 13-16 of figure 13) copies the contents of each set into the variable *citytemp* starting with the second set from the beginning. The first integer in the command *_citytemp[1:0]* (line 14 figure 13) refers to the beginning of the copying process (since the array index starts at 0, the position 1 means starting from the second element). The second integer of *_citytemp[1:0]* refers to the last index of *_citytemp[]* variable. This process will remove the first row from the table which contains the header information (lines 22- 29 of figure 14). The last command (lines 17-20 figure 13) extracts each individual cell value from *citytemp* variable and stores it in the specified variable as per line 17 (i.e. country, c_url, city, weath_today, etc.). After the five commands are executed, the variables will store the data of interest.



```
<HTML>
<HEAD>
<TITLE>INTELLICAST: europe weather</TITLE>
<A NAME="europe"></A>
<TABLE BORDER=0 CELLPADDING=0 CELLSPACING=0 WIDTH=509>
<TR>
<TD colspan=11><I>Click on a city for local forecasts</I><BR></TD>
</TR>
<TR>
<TD colspan=11><I> temperatures listed in degrees celsius </I><BR></TD>
</TR>
<TR>
<TD colspan=11><HR NOSHADE SIZE=6 WIDTH=509></TD>
</TR>
</TABLE>
<TABLE CELLSPACING=0 CELLPADDING=0 WIDTH=514>
  <TR ALIGN=left>
    <TH COLSPAN=2><BR></TH>
    <TH COLSPAN=2><I>Tue, Jan 28, 1997</I></TH>
    <TH COLSPAN=2><I>Wed, Jan 29, 1997</I></TH>
  </TR>
  <TR ALIGN=left>
    <TH><I>country</I></TH>
    <TH><I>city</I></TH>
    <TH><I>forecast</I></TH>
    <TH><I>hi/lo</I></TH>
    <TH><I>forecast</I></TH>
    <TH><I>hi/lo</I></TH>
  </TR>
  <TR ALIGN=left>
    <TD>Austria</TD>
    <TD><A HREF=http://www.intellicast.com/weather/vie/>Vienna</A></TD>
    <TD>snow</TD>
    <TD>-2/-7</TD>
    <TD>snow</TD>
    <TD>-2/-7</TD>
  </TR>
  <TR ALIGN=left>
    <TD>Belgium</TD>
    <TD><A HREF=http://www.intellicast.com/weather/bru/>Brussels</A></TD>
    <TD>fog</TD>
    <TD>2/-2</TD>
    <TD>sleet</TD>
    <TD>3/-1</TD>
  </TR>
      .
      .
    </TABLE>
      .
  </HTML>
```

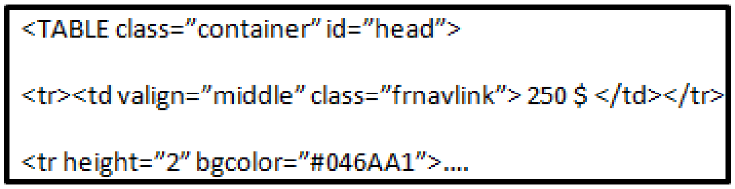
Figure 14: A section of HTML file (Hammer et al., 1997).

2.2 Supervised Wrapper

Zhai and Liu (2007) propose an instance-based learning method to extract structured data from web pages. The proposed method does not need an initial set of training pages to learn the extraction rules. Authors claim that the proposed instance-based learning method is able to start the extraction process from one manually-labeled web page. It extracts target items by comparing the prefix and suffix HTML tags of the new instance to be extracted with those of the labeled instances of targeted web page. If any item in targeted web page cannot be extracted, it is sent again to manual labeling. For example, user is interested in extracting four attributes from a product web page: name, image, description, and price. The template T for a labeled page is represented as follows:

$$T = < \text{pattern}_{\text{name}}, \text{pattern}_{\text{image}}, \text{pattern}_{\text{description}}, \text{pattern}_{\text{price}} >$$

Each pattern$_i$ in T consists of a ‘prefix’ tokens stream and ‘suffix’ tokens stream. For example, if the price of the product is embedded in the HTML source code as shown in the following code segment:

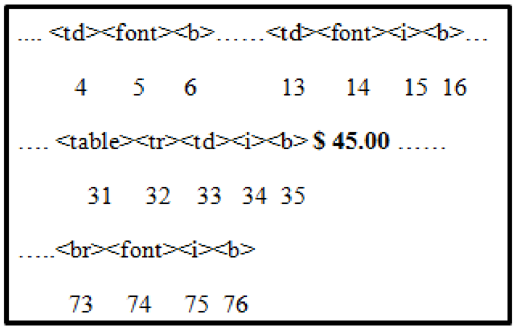


```
<TABLE class="container" id="head">

<tr><td valign="middle" class="frnavlink"> 250 $ </td></tr>

<tr height="2" bgcolor="#046AA1">....
```

Then the instance-based learning method will use the following pattern to uniquely identify the price attribute:

price pattern = (prefix: (<table><tr><td>), suffix: (</td></tr><tr>)).

For a new page d, the method compares the stored prefix and suffix strings with the tag stream of each attribute of page d. The key idea of the extraction process is based on whether the new attribute can be uniquely identified using minimum number of prefix and suffix matches which is called ‘*sufficient match*’ technique by (Zhai and Liu, 2007). If any attribute cannot be uniquely identified, page d is sent for manual labeling. For example, assume the following five HTM tokens <table> <tr> <td> <i> <b> are saved as prefix string of *price* attribute, and the HTML source code of the targeted page is given as shown in figure 15. There are four strings are matched with the prefix of *price* attribute in the four rows of the table below the first row as shown figure 15. The number inside the brackets ( ) refers to token id and the dash ‘- ‘ means that there is no match. The best match score is 5 which represents exact match with the prefix string, but the proposed method uses the sufficient match score which guarantees the uniquely identification of the attribute. In this example, the sufficient match score is 3, which comes from the match of the string <td>(33)<i>(34)><b>(35) at the fourth row of the table in figure 15. This match can uniquely identify the *price* attribute, so additional tokens <table>(31) and <tr>(32) are not needed any more. As shown in figure 15 there are four <b> strings, and three <i><b> strings together which are not sufficient for uniquely identification of price item.



```
.... <td><font><b>......<td><font><i><b>...

      4    5    6          13     14    15  16

.... <table><tr><td><i><b> $ 45.00 ......

        31    32    33   34  35

.....<br><font><i><b>

      73    74    75  76
```

Prefix :	<table>	<tr>	<td>	<i>	<b>	price
	-	-	-	-	<b>(6)	
	-	-	-	<i>(15)	<b>(16)	
	<table>(31)	<tr>(32)	<td>(33)	<i>(34)	<b>(35)	
	-	-	-	<i>(75)	<b>(76)	

Figure 15: HTML source code of web page d.

Wrapper induction is built using a learning process of the extraction rules from the initial training sets, and then applies these rules to extract web contents from other web documents. Wrapper induction can be supervised or semi-automatic and in both cases, it needs initial set of training web pages to learn the extraction rules of targeted items inside web pages. Normally user labels or marks the targeted items of the initial training web pages. Once the wrapper is learned, the extraction rules are applied for the similar targeted web pages to extract web contents and data items from these pages. Many of research works have been conducted to induct the wrapper from the initial set of training web pages such as HLRT(Kushmerick et al., 1997), SoftMealy(Hsu and Dung, 1998), STALKER (Muslea et al., 1999), WHISK(Soderland, 1999), and IEPAD (Change et al., 2001).

2.2.1 STALKER

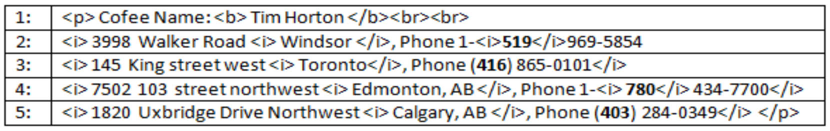
Muslea et al (1999) propose an inductive algorithm called STALKER based on the idea of hierarchal information extraction to generate high accuracy extraction rules. STALKER uses supervised learning approach to learn the extraction rules from manual-labeled training web pages. Writing extraction rules of training web pages is the main problem with wrapper induction systems. The problems of manual writing extraction rules are time consuming, need many of expertise people and many of maintenance as long as the web pages keep changing. To extract data items from web page, STALKER takes extraction rules include *Start Rule* and *End Rule* as input and starts extracting the data items. *Start Rule* represents the starting point of the data extraction and the *End Rule* represents the stopping point of extraction data. An example of STALKER algorithm is giving below:

Consider the coffee descriptions giving in figure 16. Figure 16 (a) represents an example of *'Tim Horton'* coffee training web page having four different branches in Canada. This page shows the name of the coffee in line 1 and then followed by four branches shown in lines 2-5 showing their addresses. Figure 16 (b) shows the hierarchal tree presentation of the training web page. The user wants to extract the area code of the phone numbers from all branches addresses of the coffee. The wrapper needs to go through all the following steps to do the extraction:

1. Identify the name of the coffee. Wrapper can use the start rule SkipTo (<b>), and the end rule (</b>) at line 1 to identify the name of the coffee.
2. Identify the list of addresses. Wrapper can use the start rule SkipTo (<br><br>), and the end rule </p>.
3. Iterate through the list of addresses line 2-5 to break it into four different records. To identify the beginning of each address, wrapper can use the start rule SkipTo (<i>), and end rule SkipTo (</i>).

4. Once each address is specified, wrapper needs rules to extract the area code. The following are possible rules wrapper can use to start the extraction process.

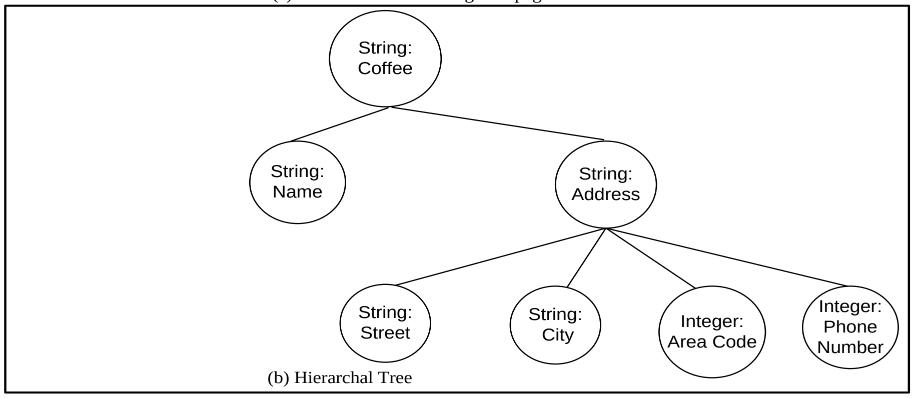
Start Rules:	End Rules:
R1: SkipTo (**(**)	SkipTo (**)**)
R2: SkipTo (**-<i>**)	SkipTo (**</i>**)



1:	
	```
	<p> Cofee Name: <b> Tim Horton </b>
	```

2:	`<i> 3998 Walker Road <i> Windsor </i>, Phone 1-<i>519</i>969-5854`
3:	`<i> 145 King street west <i> Toronto</i>, Phone (416) 865-0101</i>`
4:	`<i> 7502 103 street northwest <i> Edmonton, AB </i>, Phone 1-<i> 780</i> 434-7700</i>`
5:	`<i> 1820 Uxbridge Drive Northwest <i> Calgary, AB </i>, Phone (403) 284-0349</i> </p>`

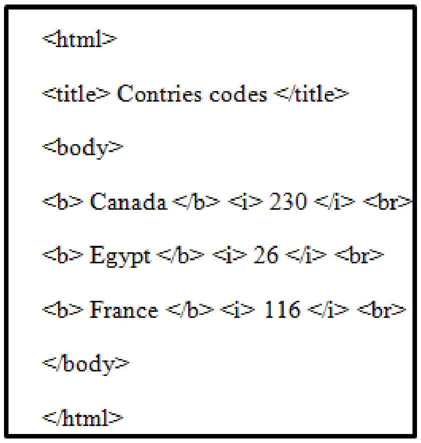
(a) HTML code of training web page



(b) Hierarchal Tree

Figure 16: (a) HTML code of training web page, (b) Tree hierarchal.

Change et al (2001) propose a new system called Information Extraction based on Pattern Discovery (IEPAD) to extract data from web documents. It is based on the fact that if the web page contains structured data records to be extracted, they are often represented using the same template. Thus, the records patterns can be discovered and the data records can be easily extracted. IEPAD automatically discovers extraction rules by identifying data records boundaries and repetitive patterns. IEPAD uses data structure called PAT tree (Morrison, 1968) to discover repetitive patterns in the web page. Once the extraction rules are discovered, the extractor receives the web page and targets patterns as input and applies pattern matching algorithm to identify and extract all the occurrences of target items. For example, in the following HTML code of a web page shown in figure 17 contains repeating pattern which can be used as input to IEPAD.



```
<html>
<title> Contries codes </title>
<body>
<b> Canada </b> <i> 230 </i> <br>
<b> Egypt </b> <i> 26 </i> <br>
<b> France </b> <i> 116 </i> <br>
</body>
</html>
```

Figure 17: Repetitive patterns.

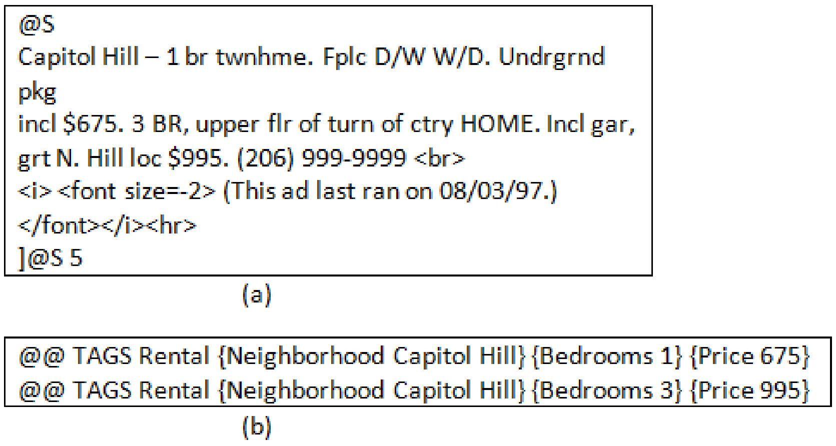
By coding each HTML tag as individual token e.g., (<b>) , and each text as (_), IEPAD generates the following pattern for the three tags occurrences in figure 17:

“ (<b>)(_) (</b>) (<i>) (_) (</i>) (<br>) “

The user has to determine which text tokens, for example, the second, and fifth (_) are important and worth to be extracted. As shown in figure 17, the second (_) text token represents the name of the country, and the fifth (_) token represents the country code.

2.2.2 WHISK

Soderland (1999) proposed a wrapper induction system called WHISK which automatically learns the extraction rules from the initial training set, and then applies these rules to extract the data from the web pages similar to those pages in training sets. WHISK is supervised wrapper that learns the extraction rules from the hand-tagged web pages. For example, figure 18 (a) shows an instance code of rental advertisement domain web page. Figure 18 (b) shows the hand-tagged instances to be extracted.



```
@S
Capitol Hill – 1 br twnhme. Fplc D/W W/D. Undrgrnd
pkg
incl $675. 3 BR, upper flr of turn of ctry HOME. Incl gar,
grt N. Hill loc $995. (206) 999-9999 <br>
<i> <font size=-2> (This ad last ran on 08/03/97.)
</font></i><hr>
]@S 5
```

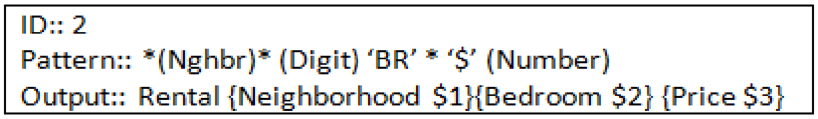
(a)

```
@@ TAGS Rental {Neighborhood Capitol Hill} {Bedrooms 1} {Price 675}
@@ TAGS Rental {Neighborhood Capitol Hill} {Bedrooms 3} {Price 995}
```

(b)

Figure 18: A training instance hand-tagged with two extractions (Soderland, 1999).

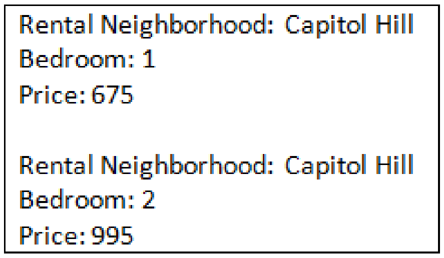
WHISK builds the extraction rules based on the training instances, so the generated extraction rules based on the hand-tagged instances in figure 18 (b) will be as shown in figure 19.



```
ID:: 2
Pattern:: *(Nghbr)* (Digit) 'BR' * '$' (Number)
Output:: Rental {Neighborhood $1}{Bedroom $2} {Price $3}
```

Figure 19: WHISK rule (Soderland, 1999).

This rules looks for the bedroom number and the associated price. The wildcard '*' means skip any sequence of characters until the next pattern. In figure 18 (a) the wrapper skips the characters until reaches the string 'Capitol Hill' and stores it in the variable $1, then it skips the code until reaches the first digit which equals to 1 and store it in the variable $2, then it continues skipping the characters until hits the first number which represents the price and equals to 675 and stores it in the variable $3. The final output of applying the WHISK rule (figure 19) on the HTML code shown in figure 18 (a) will be as shown in figure 20.


Rental Neighborhood: Capitol Hill
Bedroom: 1
Price: 675

Rental Neighborhood: Capitol Hill
Bedroom: 2
Price: 995

Figure 20: Output of wrapper WHISK.

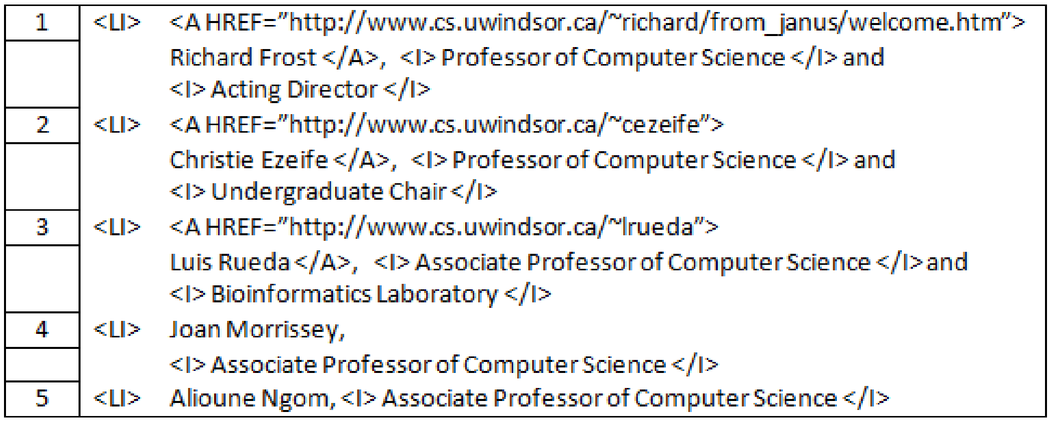
2.2.3 SoftMealy

Hsu and Dung (1998) address the problems that are related to the current existed wrappers systems. They claim that all the current existed web contents wrappers are restrictive to process the web pages that contain structured data records with missing attributes, extra attributes, and different order attributes. The authors propose a wrapper induction system called '*SoftMealy*' to extract data tuples from web pages. *SoftMealy* is based on the contextual rule concept, where each distinct attribute of each data tuple inside the web page can be represented as a transition rule. Before going to the detailed example describes how SoftMealy works, we need to introduce some definitions which have been proposed by (Hsu and Dung, 1998). They consider the HTML page as a set of tokens, and each token is denoted as the symbol *t(v)*, where *t* is a token class and *v* is a string. The following are some classes and their strings:

- CAlph(WINDSOR): All uppercase string : e.g. "WINDSOR".
- C1Alph(Professor): The first uppercase letter, followed by a string with at least one lowercase letter: e.g. "Professor".
- 0Alp(and): The first lowercase letter, followed by a string with zero or more letters: e.g. "and".
- Num(123): Numeric string: e.g. "123".
- Html(<I>): HTML tag: e.g. "<I>".
- Punc(,): punctuation symbol: e.g. ",".

- Control characters: e.g. new line “NL(1)”, e.g. four tabs “Tab(4)”, and three blank spaces “Spc(3)”.

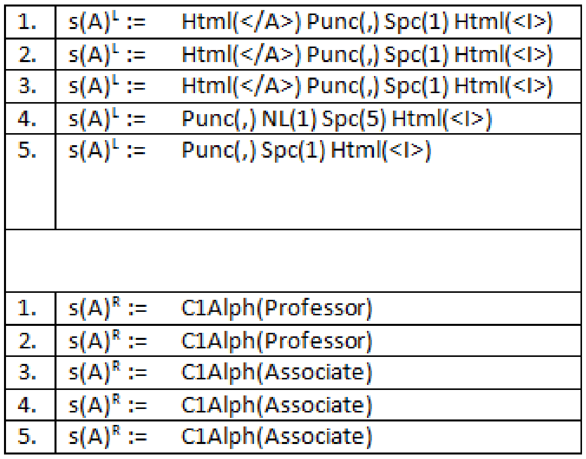
For example, if SoftMealy wrapper wants to extract structured data tuples for each faculty member in the computer science department at university of windsor through the fragmentation HTML code of the CS department web site shown in figure 21.



1	<LI>	<A HREF=”http://www.cs.uwindsor.ca/~richard/from_janus/welcome.htm”> Richard Frost </A>, <I> Professor of Computer Science </I> and <I> Acting Director </I>
2	<LI>	<A HREF=”http://www.cs.uwindsor.ca/~cezeife”> Christie Ezeife </A>, <I> Professor of Computer Science </I> and <I> Undergraduate Chair </I>
3	<LI>	<A HREF=”http://www.cs.uwindsor.ca/~lrueda”> Luis Rueda </A>, <I> Associate Professor of Computer Science </I> and <I> Bioinformatics Laboratory </I>
4	<LI>	Joan Morrissey, <I> Associate Professor of Computer Science </I>
5	<LI>	Alioune Ngom, <I> Associate Professor of Computer Science </I>

Figure 21: Fragment of HTML code for faculty member’s web page.

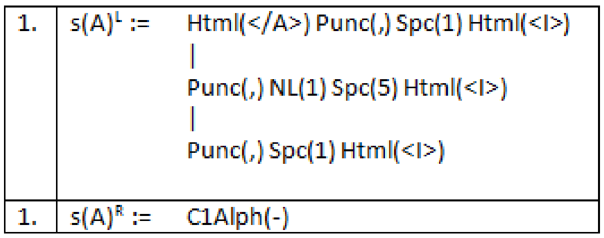
As shown in figure 21 that the HTML code contains five tuples. Each tuple provides information about the faculty member as a sequence of attributes. In this example, each data tuple contains the following attributes: URL *U*, name *N*, academic title *A*, and administrative title *M*. The data tuple can be represented as a set of attributes (*U*,*N*,*A*,*M*). Let us suppose that we need to extract the academic title *A* for each faculty member. *SoftMealy* will generate the following extraction rules (figure 22):



1.	$s(A)^L$:=	Html(</A>) Punc(,) Spc(1) Html(<I>)
2.	$s(A)^L$:=	Html(</A>) Punc(,) Spc(1) Html(<I>)
3.	$s(A)^L$:=	Html(</A>) Punc(,) Spc(1) Html(<I>)
4.	$s(A)^L$:=	Punc(,) NL(1) Spc(5) Html(<I>)
5.	$s(A)^L$:=	Punc(,) Spc(1) Html(<I>)
1.	$s(A)^R$:=	C1Alph(Professor)
2.	$s(A)^R$:=	C1Alph(Professor)
3.	$s(A)^R$:=	C1Alph(Associate)
4.	$s(A)^R$:=	C1Alph(Associate)
5.	$s(A)^R$:=	C1Alph(Associate)

Figure 22: SoftMealy extraction rules.

The first row of figure 22 represents the left separator $s(A)^L$ of attribute *A*. It states that the left context is HTML token "</A>" followed by a comma "," and one space and HTML token "<I>" at the end from left to right. While the right separator $s(A)^R$ of attribute *A* states that the right context is the string "Professor". By applying the first rule, *SoftMealy* will be able to extract all the title academic attributes *A* for the first three data tuples shown in figure 21. The wrapper will start from left to right parsing the HTML tokens of data tuple number 1 of figure 21 and will finds the matched tokens "</A> , <I> " which represents the left delimiter of the attribute *A*. Then the wrapper continues in parsing the HTML tokens until finds the string "Professor" which represents the right delimiter of the attribute *A*. In this case the *SoftMealy* knows the boundary of the academic title attribute *A* and be able to extract it for each data tuple. It discovers that there are some repetitions in the extraction rules, so it generalises them as shown in figure 23. Where "|" means "or". SoftMealy covers the delimiters whose left context syntax matches one of three distinct $s(A)^L$ in figure 23.



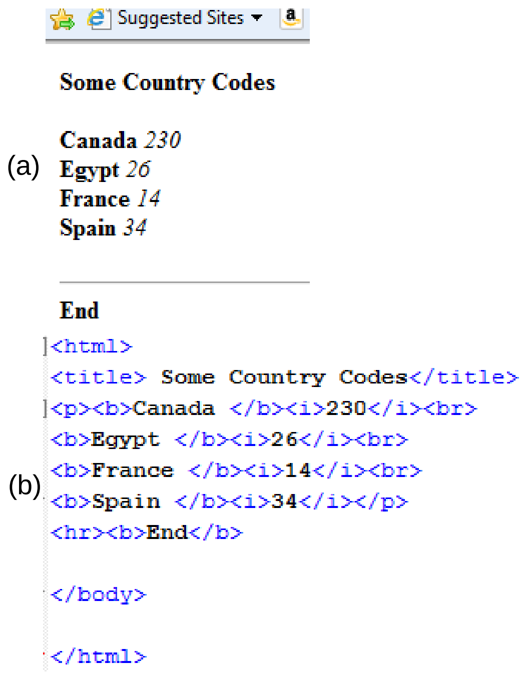
1.	s(A)^L :=	Html(</A>) Punc(,) Spc(1) Html(<I>)
		|
		Punc(,) NL(1) Spc(5) Html(<I>)
		|
		Punc(,) Spc(1) Html(<I>)

1.	s(A)^R :=	C1Alph(-)

Figure 23: SoftMealy generalised extraction rules.

2.2.4 HLRT

Kushmerick et al.(1997) address the problem of wrapper construction through query responses. For example, let us suppose there is a web page browses a tabular list of countries and their phone country codes, and a user got an answer for a query as shown in figure 24(a). The HTML code of the web page is rendered in figure 24(b).



Some Country Codes

Canada *230*
Egypt *26*
France *14*
Spain *34*

End

```
<html>
<title> Some Country Codes</title>
<p><b>Canada </b><i>230</i><br>
<b>Egypt </b><i>26</i><br>
<b>France </b><i>14</i><br>
<b>Spain </b><i>34</i></p>
<hr><b>End</b>

</body>

</html>
```

Figure 24: (a) Example query response web page; (b) The HTML code of the page.

The authors consider the wrapper uses the string position as a delimiter between the extracted contents. It is obviously in figure 24 (b) that the country's names are surrounded by <b> and </b> tokens, and the country codes by <i> and </i>. This is called left-right (*LR*) strategy to extract the required web contents. But *LR* fails in extracting many of web contents because not all the contents between tags <b> and </b> are country names. For example, the last 'End' string of figure 24(b) is surrounded by <b> and </b> but it is not country name, and *LR* extracts it as a country name. For that reasons, (Kushmerick et al., 1997) propose to use the *ExecuteHLRT* strategy to distinguish the beginning of the head of the page and the last tuple inside the body of the page. *HLRT* stands for header, left, right, and tail; where it is considered a more sophisticated approach than *LR* approach. Figure 25 shows the code of the generated wrapper by *ExecuteHLRT* strategy.

```
ExecuteHLRT(<h,t,$l_1$,$r_1$,…,$l_k$,$r_k$>, page P)
    skip past first occurrence of h in P
    while next l1 is before next t in P
        For each <$l_k$,$r_k$> ∈ {<$l_1$,$r_1$>,…, <$l_k$,$r_k$>}
            Skip past next occurrence of $l_k$ in P
            Extract attributes from P to next occurrence of $r_k$
    Return extracted tuples
```

Figure 25: Generated wrapper using HLRT.

The algorithm in figure 25 skips the header of web page, and for each data tuple it skips the tuple delimiters and extracts the data contents which are surrounded by these delimiters. For the first tuple in figure 24 (b) ' *<b> Canada </b> <i> 230 </i>*', the algorithm skips the first left delimiter l_1= <b> and the first right delimiter r_1= </b> , and extracts the first attribute *k=1* which is the string 'Canada' in this example. Then it skips the second left delimiter l_2= <i> and the second right delimiter r_2= </i>, and extracts the second attribute k=2 which is the Canada's phone area code with the string value equals to '230'. Then the algorithm continues in extracting all the data tuples from the HTML code until hits the tail *t* of the web page which is the token <hr> in this example.

2.3 Automatic Wrapper Generation

Wrapper generation using supervised learning suffers from two main shortcomings: it is not applicable for large number of websites due to large efforts that are required to do the manual labeling of web pages in training data sets. For example, if a website needs to know all the products which have been sold in web, it will be kind of impossible task to manually label all these products. The cost of wrapper maintenance is considered very high. Web is a dynamic repository and keeps updating constantly. Since wrapper depends on HTML format of web page, any change in the template or code of the page will disable the work of the wrapper. So the wrapper needs constant changing and manually maintenance to repair it which make the task of build and maintain the supervised wrapper system very costly. For all the previous problems which are related to supervised wrapper generation, researchers have studied the idea of build automatic data extraction from the web, and found it possible. (Liu et al., 2003 ; Zhai and Liu, 2005; Annoni and Ezeife, 2009; Mutsuddy, 2010) proposed different automatic web contents extraction systems to extract data from web documents and store it in structured database to facilitate the data retrieval process.

2.3.1 Automatic Extraction using Tag-Tree

Liu et al. (2003) propose a new method called mining data record (*MDR*) from web pages to automatically extract structured data records from web pages and store them in relational database. Authors claim that their proposed method is based on two main observations: 1) A group of data records which contain similar data objects are normally located at a particular region, and share in common similar HTML tags in their structure template. They call such region a data region. 2) The nested structure of HTML tags can be represented as a tag tree. Figure 26 shows a tag tree of a list product web page. As shown in figure 26 that the web page contains two adjacent data records in the two dashed boxes, each data record is wrapped by five *tr* tag nodes and they have the same parent node which is *tbody*.

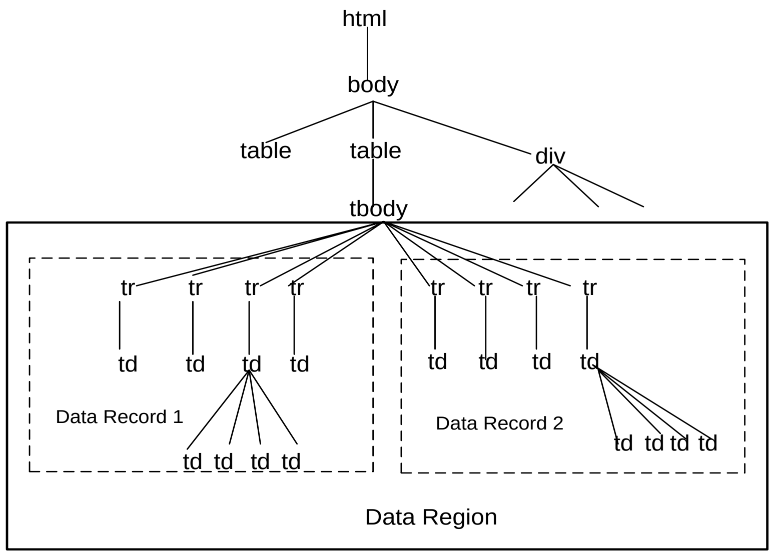


Figure 26: Tag-Tree of a list product web page.

MDR method represents each web page as a tag-tree, where each HTML tag is represented as a node such as <*tr*> tag. After building tag-tree, MDR mines each data region inside web page. (Liu et al. 2003) define a *data region* as '*a collection of two or more generalized nodes with the following properties: a) all the generalized nodes have the same parent; b) all the generalized nodes are adjacent and have the same length*'. They also define the *generalized* node as '*a node of length r (r >= 1) of HTML tags nodes of tag-tree and all the nodes are adjacent and have the same parent*'. They also consider each data record as a generalized node. As shown in figure 26 that the *data record 1* has four adjacent *tr* tag nodes, and the data region consists of two adjacent generalized nodes with the same parent *tbody* node and have equal length which equals to four. After identifying each data region in the page, MDR mine data records inside each data region and store them in structured database.

Zhai and Liu (2005) address the problem of extracting structured data records and their attributes from web pages and store them in relational database. They claim that the current existed methods do not tackle this problem so far. For this reason, they develop their algorithm *MDR* (Liu et al. 2003) to build a new system called Data Extraction based on Partial Tree Alignment (*DEPTA*). In their new proposed method partial-tree alignment, they initially pick the seed tree *Ts* to be the optimal tree which contains the maximum number of data attributes and then gradually compare *Ts* with other sub tag-trees which represent data records to grow *Ts* and get the final tag-tree which contains almost all the expected data attributes. The proposed algorithm represents each data record in the page as a sub tag-tree *Ti*, and for each *Ti*(i≠s), it matches each node in *Ti* with *Ts*. If the match is found for node *Ti*[j], a node *Ti*[j] is inserted in *Ts*. If the match is not found the sub tag-tree *Ti* is inserted in temporary next matching repository *R* for subsequent matching. Figure 27 shows the partial aligning multiple trees. The algorithm accepts the initial three sub-trees S={T_1, T_2, and T_3} as an input. It choices T_1 as *Ts* and removes T_1 from *S*. It then aligns the rest of sub-trees in *S* against *Ts* until *S* gets empty. In figure 27, it aligns T_2 to *Ts*, they produce one match in node *b*, nodes *n, c, k,* and *g* do not fit in *Ts* and there are no locations for them in *Ts*, so they could not be inserted in *Ts*. It inserts T_2 in R for holding to be matched later. Now it matches T_3 with *Ts*, it finds that the unmatched nodes *c, h,* and *k* can be inserted into *Ts*, it updates *Ts* and sets the flag to be '*true*' to indicate that the original seed tree *Ts* has been modified. It then checks for the stopping condition whether *S=0* and *flag='true'*, which means that all the sub-trees in *S* have been processed and the original seed tree has been modified. Then the sub-tress in temporary repository *R* should be processed. In figure 27, there is only one sub-tree T_2. So *T2* is matched with *Ts* and unmatched nodes *n*, and *g* are inserted to generate the final seed tree *Ts* which contains all the required attributes.

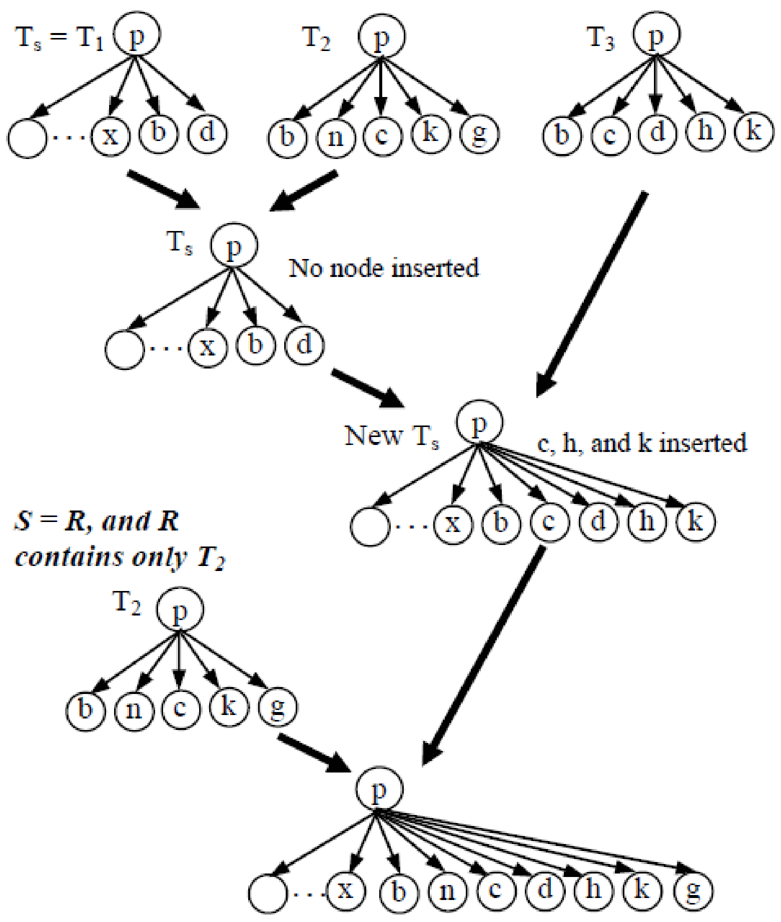
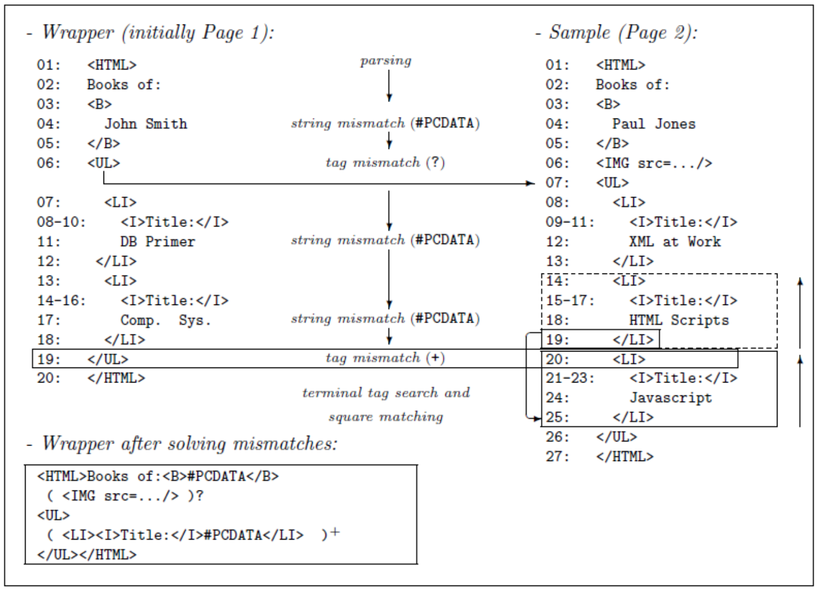


Figure 27: Iterative tree alignment with two iterations (Zhai and Liu, 2005).

Crescenzi et al. (2002) propose a new technique to automatically extract web contents from web using automatically generated wrapper. The authors tackle the problem of extracting the underlying database structure from which the structured data records inside web pages have been generated. The main motivation of their work is that the website generation process can be seen as an encoding to the original underlying database from which the data records were generated. The proposed system is called '*RoadRunner'* uses the matching technique called *ACME* which stands for Align, Collapse under Mismatch, and Extract to compare HTML pages of the same class and automatically generate wrapper based on the similarities and differences between wrapper optimal page and targeted sample page. It compares the two pages using *ACME* technique to align matches and collapse mismatches. There are two types of mismatches: string mismatches '*#PCDATA'* which happens when different strings occur in the same position between wrapper page and sample page. Tag mismatches '*(tag)?*,' happens when different HTML tag or operator occurs between the two compared pages. *RoadRunner* uses *UFRE* (Union Free Regular Expression) in building wrapper to reduce the complexity of the proposed algorithm. Figure 28 shows an example of matching wrapper page and sample page to generate wrapper. As shown in figure 28, the first string mismatch occurred at token number 4, to solve this problem *RoadRunner* generalizes the wrapper to label the discovered mismatch by replacing the string '*John Smith*' by the string '*#PCDATA*', the same thing will happen later for the string '*DB Primer*'. As shown in figure 28, the first tag mismatch occurred at token 6 of sample page, where *<IMG>* tag would be added to generalize the wrapper as optional tag. The match process continues until *RoadRunner* received to the end of one of the two compared web pages. Finally, the proposed algorithm generates wrapper which represent extraction rule that can applied to extract data from other web pages similar to wrapper page. As shown in figure 28, the generated wrapper is located at the bottom left corner of the figure.


- Wrapper (initially Page 1):
- Sample (Page 2):
parsing
string mismatch (#PCDATA)
tag mismatch (?)
string mismatch (#PCDATA)
string mismatch (#PCDATA)
tag mismatch (+)
terminal tag search and
square matching
- Wrapper after solving mismatches:

```
01:    <HTML>
02:    Books of:
03:    <B>
04:      John Smith
05:    </B>
06:    <UL>
07:      <LI>
08-10:     <I>Title:</I>
11:        DB Primer
12:     </LI>
13:      <LI>
14-16:     <I>Title:</I>
17:        Comp.  Sys.
18:      </LI>
19:    </UL>
20:    </HTML>
```

```
01:    <HTML>
02:    Books of:
03:    <B>
04:      Paul Jones
05:    </B>
06:    <IMG src=.../>
07:    <UL>
08:      <LI>
09-11:     <I>Title:</I>
12:        XML at Work
13:      </LI>
14:      <LI>
15-17:     <I>Title:</I>
18:        HTML Scripts
19:      </LI>
20:      <LI>
21-23:     <I>Title:</I>
24:        Javascript
25:      </LI>
26:    </UL>
27:    </HTML>
```

```
<HTML>Books of:<B>#PCDATA</B>
 ( <IMG src=.../> )?
<UL>
 ( <LI><I>Title:</I>#PCDATA</LI>  )+
</UL></HTML>
```

Figure 28: RoadRunner matching (Crescenzi et al., 2002).

2.3.2 Automatic Extraction using DOM-Tree

Annoni and Ezeife (2009) claim that the current existed data extraction models failed to catch the full diversity of different types of web contents. They propose a new object-oriented model to encapsulating heterogeneous web contents into object class hierarchy to extract and mine web content in a unified way. Their paper has two major contributions for web contents extraction: 1) Build a unique object-oriented model that focuses on web document content and presentation structure. 2) Extract and mine all different types of web contents regardless of the web document structure (unstructured, strictly structured). Annoni and Ezeife (2009) present web page as a *DOM tree*, and use visual based context structure, and data presentation features to identify the hierarchy of web object model. They define web page as a web zone object which is a composition of WebElement objects, and WebRender objects. The WebZone is divided into three sub-zones header, body, and footer zone. They define an algorithm which takes HTML page as input and returns the web zone objects, where they suppose that any web document contains three zones at most. So the number of series which are used to separate these zones equals to two at most. They define *series1* to separate header zone on body zone, and define *series2* to separate body zone on footer zone.

They classify web content objects into six types by relying on four basic types which are already defined by Levering and Cutler (2006): Text, Image, Form, and Plug-in content. They add two new types: Separator element and Structure element. They also classify web presentation objects into six types: Banner, Menu, Interaction, LegalInformation, Record, and Bulk. Figure 29 shows main algorithm OWebMiner () by Annoni and Ezeife (2009).

```
Algorithm OWebMiner()
Input: a set of HTML files (WDHTMLFile) of web documents.
Output: a set of patterns of objects.
Begin
For each WDHTMLFile
        (A) Extract web presentation objects and content objects
            sequentially with respect to their hierarchical dependencies.
        (B) Store the object hierarchies into a database table
endFor
(C) Mine patterns lying within objects
end
```

Figure 29: OWebMiner () algorithm (Annoni and Ezeife , 2009).

This algorithm takes web page (WDHTMLFile) as input and returns a set of patterns of extracted objects. Line (A) of the algorithm extracts all the content and presentation objects inside web page and stores them into two separate object arrays according to their DOM tree hierarchical dependencies. Line (B) stores the objects into relational database. Line (C) mines the stored objects inside database and builds the patterns of these objects. They also develop two important sub-algorithms called PresWebObjectScan() and ContWebObjectScan(). ContWebObjectScan() stores the extracted web content objects into ContentOjectArray[]. It process DOM tree starting <HTML> node until hits *series1*, then it calls algorithm ProcessContentSibling() to start extracting of content objects until hits *series2*. The algorithm recursively traverses DOM tree *block-level* tags by depth-first search until it hits *non block-level* tag. Then it processes all its siblings by breadth-first search until hits *block-level* tag or no more siblings are left. For all the siblings of non-block tags the ProcessContentSibling() algorithm stores each extracted content object in ContentObjectArray[]. Finally, the algorithm returns ContentObjectArray[].

Example OWebMiner:

This example illustrates the extraction process flow of the proposed algorithm ProcessPresentationSibling() proposed by (Annoni and Ezeife , 2009) to extract both web presentation objects and web content objects from the given web page. ProcessPresentationSibling() algorithm receives *DOM tree* of web page as input. It returns an array contains the extracted objects that are labelled with a counter called ‘*indTag*’. It scans the *block-level* tags by depth-first search until a *non-block* level tag is found. But it follows breadth-first search when a *non-block* level tag is parsed until a block-level is found or no more siblings are left. The authors applied the ProcessPresentationSibling() algorithm on a web document from Chapters7.ca web site to extract the presentation and web content objects from that web page. Figure 30 shows the input *DOM tree* of web page of ‘Chapters.ca’ web site.

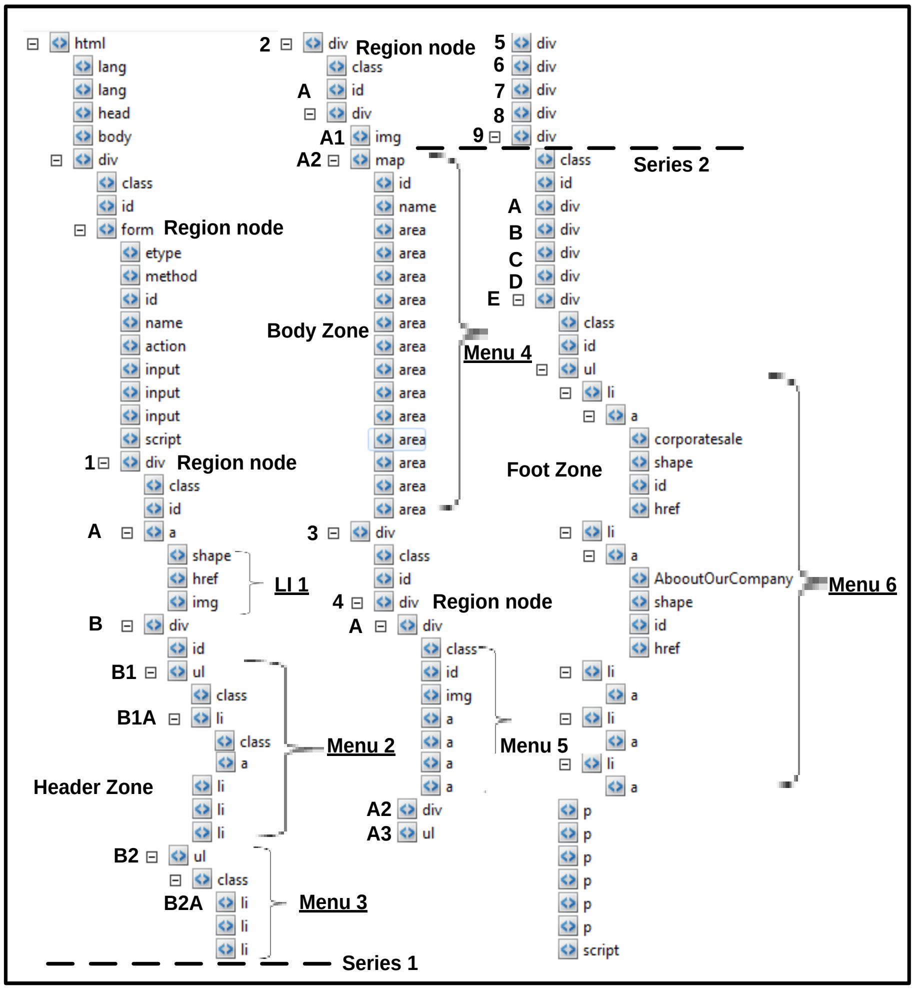


Figure 30: DOM tree of web document from Chapters.ca website.

The process begins from the *DOM tree* root ‘*<html>*’ tag, reaches *<div1>* node which is the first tag of the header zone based on the definition by the authors. Since *<div1>* is a *block-level* tag the algorithm continues in depth-first search scanning until reaches node ‘*<a>*’ which is labelled by the letter '***A***', and it continues the depth-first search until reaches '*shape*' node which is a *non-block* level node, so the '*shape*' node and all its siblings are extracted and stored into '*tagArray[0]*' which represents link object with its related image labeled by the string '**LI1**', and '*indTag*' counter is increased by 1. Then ProcessPresentationSibling() algorithm recursively is called with the *DOM tree* and the input string 'html/body/form/div1/div=B'. Since <div=B> node (labelled by the letter '***B***' figure 30) is a block-level node the algorithm continues scanning in depth-first search until reaches node '*ul*'('***B1***'), and again by the depth-first search it reaches the node '<li>' ('***B1A***') which is a non-bock level nodes so '<li>=B1A' and all its siblings which are non-block are stored in '*tagArray[1]*' which represent the second presentation menu (**Menu2**)object because the tagArray is composed of four <li> tags and each of them has inner <a> link tag, and the '*indTag*' counter is increased by one. Then the algorithm is called recursively with the input 'html/body/form/div1/div/ul=B2' since '<ul>'('***B2***') is a *block-level* node the algorithm continues in depth-first search until reaches '<li>=B2A' node ('**B2A**') which is a *non-block* level node, so '<li>=B2A' and all its sibling are extracted and stored in '*tagArray[3]*' as the third presentation menu object (**Menu 3**). The ProcessPresentationSibling() algorithm recursively keeps continue scanning and mining content objects inside given web page using depth-first search and breadth-first search until the end of the footer zone.

Mutsuddy (2010); Ezeife and Mutsuddy (2013) developed the work by (Annoni and Ezeife, 2009) to build a new system called WebOMiner. They define data block and data region to ensure the consistency and the relationship between related data items which is the case that was not tackled by (Annoni and Ezeife, 2009). So they modified ProcessContentSibling() algorithm to identify data block and data region. Also they address a new idea which is based on relating HTML tags with their attributes information to uniquely identify each web content type. They propose a full object-oriented model which consists of four modules to extract all types of data objects inside web page regardless of their structured formats: A) Crawler Module; B) Cleaner Module; C) Extractor Module; and D) Miner Module. Figure 31 shows the proposed architecture of web miner model by (Mutsuddy, 2010; Ezeife and Mutsuddy, 2013).

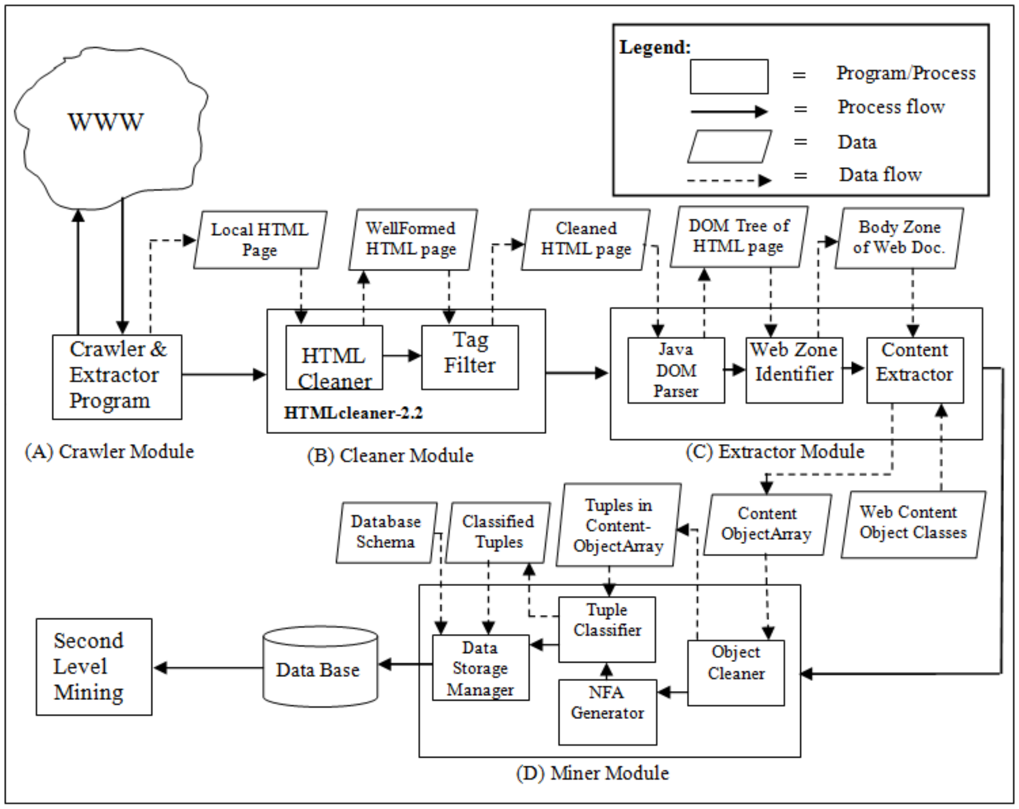


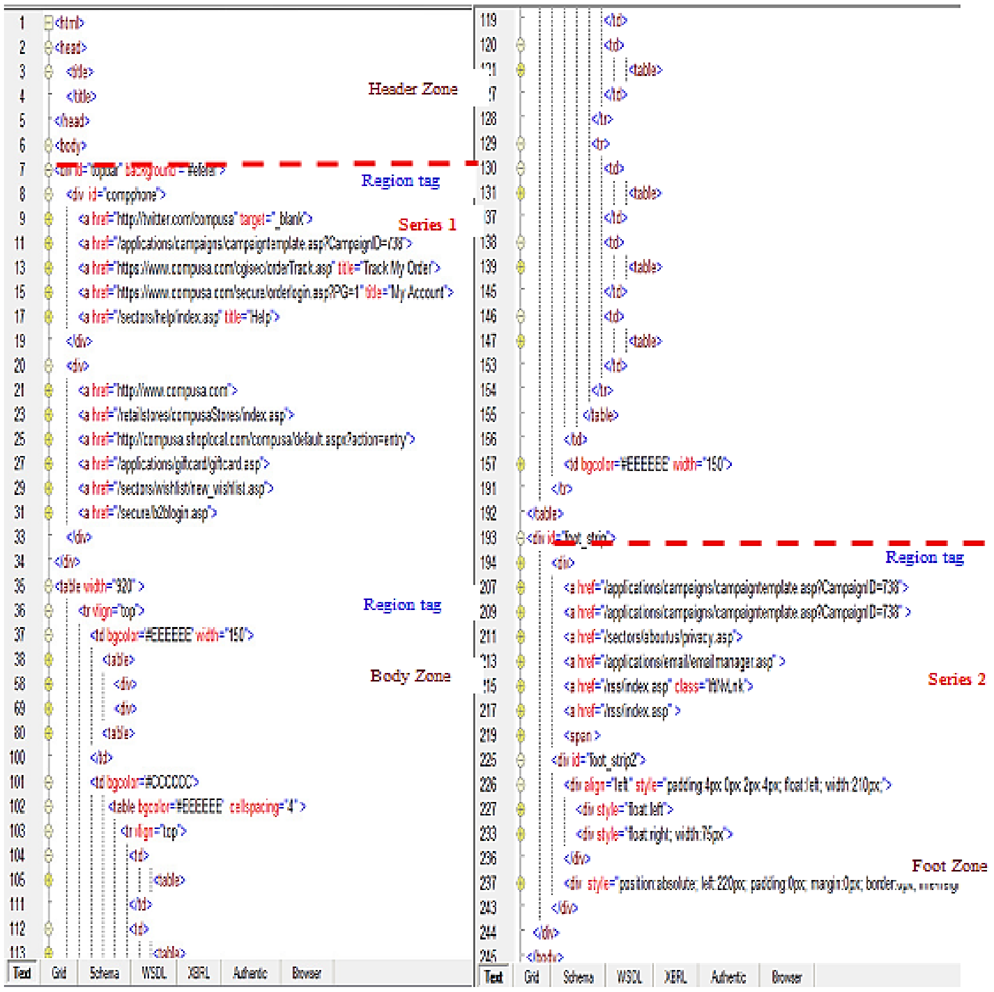
Figure 31: WebOMiner Architecture Model (Ezeife and Mutsuddy, 2013).

The authors developed a mini-crawler algorithm that takes URL of targeted list product web page as input, and downloads the page from the web and stores it in the local computer for further processing. The cleaner module uses HTML-Cleaner 2.2 to clean the downloaded web page from noise HTML tags and format scripts. The HTML-Cleaner 2.2 software is an open source and can be downloaded from web for free. The extractor module builds *DOM Tree* of the web page, and then applies the modified extraction algorithm ProcessContentSibling() by (Annoni and Ezeife, 2009) to extracted the data objects from the page and store them in ContentObjectArray[] array. The miner module mines and labels the extracted data objects in ContentObjectArray[] based on the appropriate defined class of the object. Mutsuddy (2010); Ezeife and Mutsuddy (2013) categorise data objects into six types: product, list, link, text, form, noise. They build none full automatic structure of Non-deterministic Finite Automata (NFA) for each type of the six objects based on their observations for the contents of web pages which are in the domain of the B2C websites. The WebOminer() algorithm in miner module scans ContentObjectArray[] array for the second time to classify each data object based on the matched structured NFA.

Example WebOMiner:

Mutsuddy (2010); Ezeife and Mutsuddy (2013) modified ProcessContentSibling() algorithm by (Annoni and Ezeife, 2009). This application example shows how the modified version of ProcessContentSibling() algorithm extracts the data objects from the *DOM tree* of the web list product page shown in figure 32 which has been used as a running example by (Mutsuddy, 2010). The DOM tree will be built automatically by importing the required DOM tree packages, and calling special classes' functions like DocumentBuilderFactory, and NodeIterator in Java programming language. This algorithm extracts data objects from *DOM tree* and stores them in ContentObjectArray[] array until it hits the Foot zone which is labelled by '***series-2***' (figure 32). In this running example, '***series-1***' is assigned by *<div>* tag at line 7 (figure 32) which is a block level tag, so the algorithm calls the CheckTagObject() algorithm which creates an *OpenSeparator* object '{' and stores it in the first position of ContentObjectArray[0], then *TTag* variable is set to the next child tag '*<div>*' at line 8 (figure 32) and the same thing will occur that CheckTagObject() algorithm stores another *OpenSeparator* object '{' it in ContentObjectArray[1]. Now the TTag variable is set to the next child node '*<a>*' at line 9 and the algorithm calls itself recursively. Since '*<a>*' is a *non-block* level tag the algorithms extracts the five sequence siblings (line 9 to 17 figure 32) and stores the respective '*<link>*'

followed by ‘<*image*>’ objects for all the five siblings into ContentObjectArray[2]-ContentObjectArray[11]. Line 19 ends the data block with the ‘</*div*>’ tag and the algorithm stores the CloseSeparator object ‘}’ into ContentObjectArray[12]. Figure 33 shows a snapshot of the ContentObjectArray[] array after storing the web content objects inside it.



```
1    <html>
2    <head>
3      <title>
4      </title>
5    </head>
6    <body>
7    <div id="topbar" background="#efefef">
8      <div id="compphone">
9        <a href="http://twitter.com/compusa" target="_blank">
11       <a href="/applications/campaigns/campaigntemplate.asp?CampaignID=738">
13       <a href="https://www.compusa.com/cgisec/orderTrack.asp" title="Track My Order">
15       <a href="https://www.compusa.com/secure/order/login.asp?PG=1" title="My Account">
17       <a href="/sectors/help/index.asp" title="Help">
19     </div>
20     <div>
21       <a href="http://www.compusa.com">
23       <a href="/retailstores/compusaStores/index.asp">
25       <a href="http://compusa.shoplocal.com/compusa/default.aspx?action=entry">
27       <a href="/applications/giftcard/giftcard.asp">
29       <a href="/sectors/wishlist/new_wishlist.asp">
31       <a href="/secure/b2blogin.asp">
33     </div>
34   </div>
35   <table width="920" >
36     <tr valign="top">
37       <td bgcolor="#EEEEEE" width="150">
38         <table>
58           <div>
69           <div>
80         <table>
100      </td>
101      <td bgcolor="#CCCCCC">
102        <table bgcolor="#EEEEEE" cellspacing="4">
103          <tr valign="top">
104            <td>
106              <table>
111            </td>
112            <td>
113              <table>
```

```
119            </td>
120            <td>
121              <table>
127            </td>
128          </tr>
129          <tr>
130            <td>
131              <table>
137            </td>
138            <td>
139              <table>
145            </td>
146            <td>
147              <table>
153            </td>
154          </tr>
155        </table>
156      </td>
157      <td bgcolor="#EEEEEE" width="150">
191    </tr>
192  </table>
193  <div id="foot_strip">
194    <div>
207      <a href="/applications/campaigns/campaigntemplate.asp?CampaignID=738">
209      <a href="/applications/campaigns/campaigntemplate.asp?CampaignID=738" >
211      <a href="/sectors/aboutus/privacy.asp">
213      <a href="/applications/email/emailmanager.asp" >
215      <a href="/rss/index.asp" class="RSSLink">
217      <a href="/rss/index.asp" >
219      <span >
225    <div id="foot_strip2">
226      <div align="left" style="padding:4px 0px 2px 4px; float:left; width:210px;">
227        <div style="float:left">
233        <div style="float:right; width:75px">
236      </div>
237      <div style="position:absolute; left:220px; padding:0px; margin:0px; border:
243    </div>
244  </div>
245  </body>
```

Figure 32: DOM tree of the product list web page.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
{	{	<ink>	<img>	<link>	<img>	<link>	<img>	<link>	<img>	<link>	<img>	}		

Figure 33: Sample of ContentObjectArray[] array.

Harunorrashid (2012) develops the WebOMiner to a new version called the WebOMiner-2. The new miner system tackles the manual process of generating the NFA structure of the data blocks which is considered the main shortcoming of WebOMiner. The WebOMiner-2 automates the process of generating the NFA structure of data blocks through building the regular expression of frequent patterns of data attributes which are extracted from DOM-Tree and stored in contentObjectArray[]. The generated regular expressions will be input for the NFA generator algorithm which builds the NFA structures as shown in figure 34. The WebOMiner-2 generates the database schema automatically to store different types of web content objects (list, product, text). The WebOMiner-2 suffers from some shortcomings, it still uses the concepts of block level and non-block level data blocks which are proposed by (Annoni and Ezeife, 2009). Identifying the data block and data region by WebOMiner-2 is very costly, where it requires scanning each tag node in DOM-Tree to recognise whether it is a block level or non-block level data block. Also the WebOMiner-2 stores the extracted data tuples in relational database (RDB) which neither historical nor object-oriented database.

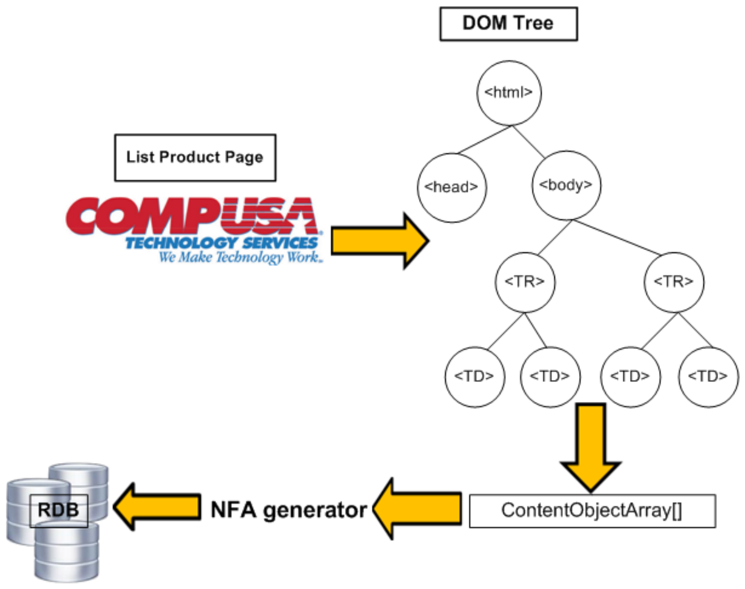


Figure 34: workflow of WebOMiner-2 (Harunorrashid, 2012)

CHAPTER 3 - Advanced Full Automatic WebOMiner-3

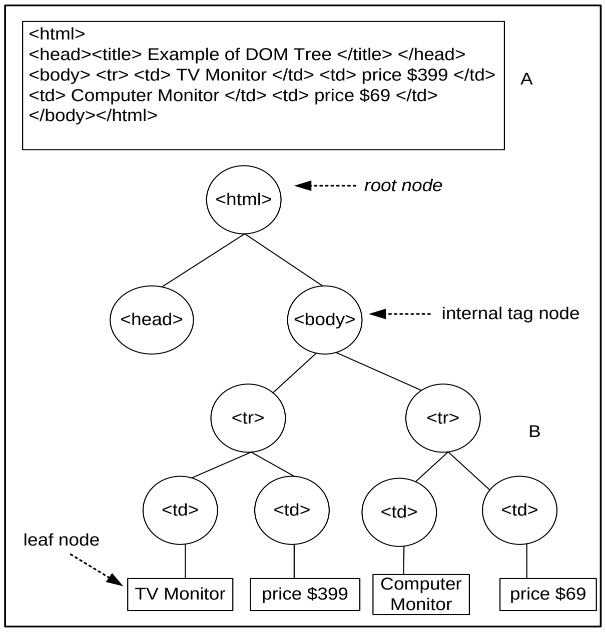
As we discussed in section 2.3.2, Mutsuddy (2010); Ezeife and Mutsuddy (2013) proposed a new complete model called Web Object Miner (WebOMiner), and Harunorrashid (2012) develops it to WebOMiner-2 to extract and mine data objects from the list product web pages in the domain of B2C websites. Their proposed systems suffer from some shortcomings and needs to be advanced. We studied their work and propose a new miner module called WebOMiner-3. The new version of miner module generates the schema for more than one website and does the integration between different schemas to build a unique object-oriented data warehouse (OODWH) to store the extracted data from different B2C websites such as ‘compUSA’, ‘BestBuy’. The WebOMiner-3 proposes a new extraction algorithm based on the similarity of the DOM tree tag nodes of the targeted data blocks; builds a full automatic NFA structure for each data tuple type based on frequent object mining (FOM) technique which is also proposed in this thesis. Finally build object-oriented automatic data warehouse to integrate and store historical information about different product items from web sources. This thesis addresses the shortcomings of the works by Mutsuddy (2010) ; Harunorrashid (2012); Ezeife and Mutsuddy (2013) and tackle them, and advance the WebOMiner and WebOMiner-2 systems. Our approach is deeply discussed in the subsequent sections of this chapter: section 3.1 introduces some preliminaries definitions that will be used in the rest of this thesis; section 3.2 presents the overall architecture of the advanced WebOMiner-3 model. Section 3.3 describes the OO data warehouse and integration module, the complexity analysis of the system is described in section 3.4.

3.1 Preliminaries

In this section, we introduce some definitions and concepts that will help to deeply understand the target of this thesis and move on understanding the proposed new advanced WebOMiner-3 model.

3.1.1 DOM Tree

The Document Object Model (DOM) tree is an application programming interface (API) that represents valid HTML page, and formatted XML page as a tree logical data structure. Each document contains only one root node which represents the top hierarchy of the page, and internal nodes which represent the HTML or XML tags, and the leaf nodes which represent the last level of the DOM tree and normally contains text data type. Figure 35 (A) shows a snapshot of HTML code for a list product web page. Figure 35 (B) shows the DOM tree of the web page in figure 35(A).



```
<html>
<head><title> Example of DOM Tree </title> </head>
<body> <tr> <td> TV Monitor </td> <td> price $399 </td>
<td> Computer Monitor </td> <td> price $69 </td>
</body></html>
```

Figure 35: (A) snapshot of HTML code for list product web page (B) Dom tree.

3.1.2 Data Block and Data Region

- ***Data Block (tuple):*** *a sub-tree of DOM tree of n (n>1) tag nodes which are siblings and have the same parent tag node.*
- ***Data Region:*** *is a group of one or more similar data blocks (d>=1) which are siblings and have the same parent tag node.*

The definition of data region can be reformulated as *if T represents the DOM tree structure of product list page p_i, and $\Re_1, \Re_2, \Re_3, ..., \Re_n$ represent data regions in T, then $\Re_i$ is a sub-tree of T and $\forall \Re_i \in T$ are siblings. A data region $\Re_i$ contains a set of adjacent similar data blocks τ which share the same parent tag node. All data blocks τ in one data region $\Re_i$ have the same number and similar pattern of HTML tag nodes.*

The definition of data block can be reformulated as *data block $\tau \in \Re_i$ contains a combination of more than one tag nodes which share the same parent tag node and configure a sub-tree t of main DOM Tree T*. Figure 35 shows a DOM tree to illustrate the definitions of both data block and data region. As shown in the figure that the *data block 2* which is referred to by label (r_2) contains five sibling tag nodes and all of them share the same parent node which has the label '*<table> (r_2)*'. Also figure 36 shows the outer data region which is surrounded by dashed-line rectangle contains two sibling data blocks (r_2) and (r_3), where the two data blocks have the same HTML tag pattern and the same length of tag nodes. Each data block contains the HTML tag pattern '<tr> <td> <tr> <td> <td> <td> <tr> <td> <td> <tr> <td>' and has the same length of tag nodes which equals to twelve in figure 35.

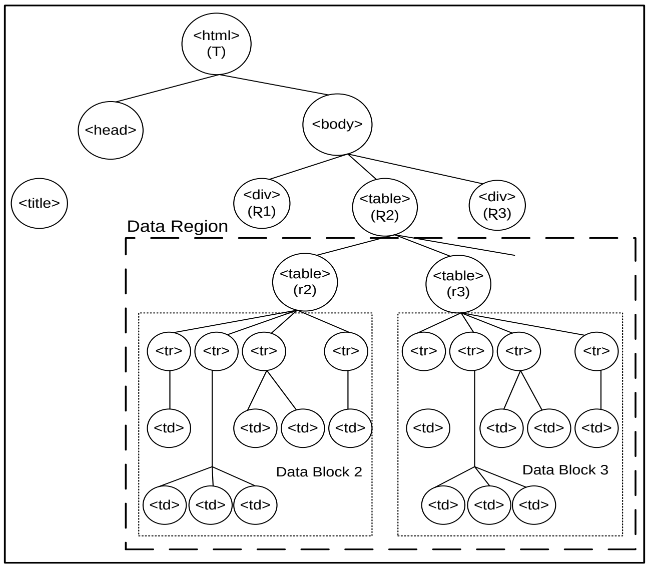


Figure 36: Graphical DOM tree illustrates data region and data block.

3.1.3 Tuple Formation from Data Block

Annoni and Ezeife (2009) categorise data blocks inside product list web page into six basic types: 1) Product data block, 2) List or Navigation data block, 3) Form data block, 4) Text Data Block, 5) Decorative/Singleton data block, and 6) Noise data block. One of Our main purposes is to fully automate the extracting process and enhance its efficiency, mining such types of data records from list and detailed products web pages. Build the historical object-oriented data warehouse repository to integrate and store the data records. The *product data block* is considered the most important data block in product list page because it contains all the information that are related to one product data tuple. Related objects of the product data block are: ‘*name*’ or ‘*title*’ of the product, ‘*image*’ of the product, ‘*product number*’, ‘*product brand*’ or ‘*type*’, ‘*product info*‘, ‘*price*’, ‘*product save*’, and additional information such as another ‘*description*’ of the product, ‘*discount*’ or if there is ‘*sale*’ on the product or not. These objects are found ordered or un-ordered in a list of flat or nested HTML tags. Each product has different structure in different list product web page. The format of the *product data block* in nested HTML tags is shown below:

{<img>, {<title>,<number>,…, <brand>,<price>}}

Ezeife and Mutsuddy (2013) propose the *separator object* to identify and separate data blocks on each other. Figure 37 shows the content objects of a *product data block* which contains a *separator object*.

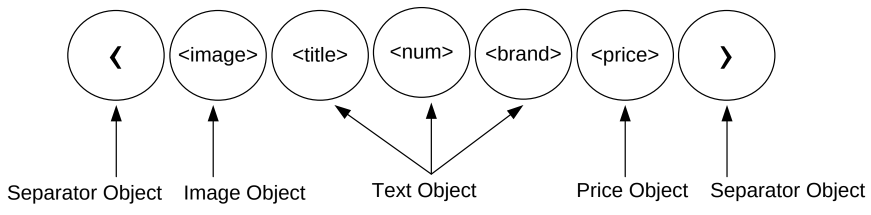


Figure 37: Objects of product data block.

3.2 Advanced Web Object Miner (WebOMiner-3)

We propose to advance '*WebOMiner*' system for the extraction and mining structured web data contents from the web using object-oriented model. Our development includes proposing a new extraction technique based on the similarity between the DOM tree tag nodes patterns. The new extraction algorithm identifies each data block individually and each data region which contains adjacent one or more data blocks. It is more efficient than the previous extraction algorithm by (Mutsuddy (2010); Ezeife and Mutsuddy (2013)) because it compares lower number of DOM tree tag nodes. Also we propose to build a full automatic NFA structure for each data tuple type to identify types of the extracted data blocks. The new automatic NFA will be generated based on the concept of frequent object mining (FOM). The new modifications include build historical object-oriented data warehouse (OODWH) to integrate and store data tuples from different list and detailed product web pages into one central data repository. The extracted data contents can be used for further mining purposes and other manipulations. Figure 38 shows the architecture of the new advanced WebOMiner-3 system. WebOMiner-3 consists of five modules: (1) Crawler module (2) Cleaner module (3) Extractor module (4) Frequent Objects Mining module (5) Data Warehouse and Integration module. These modules are called sequentially by the main algorithm WebOMiner-3 (shown in figure 39). We will explain below each module of WebOMiner-3 system individually and will deeply discuss how each algorithm of each module works.

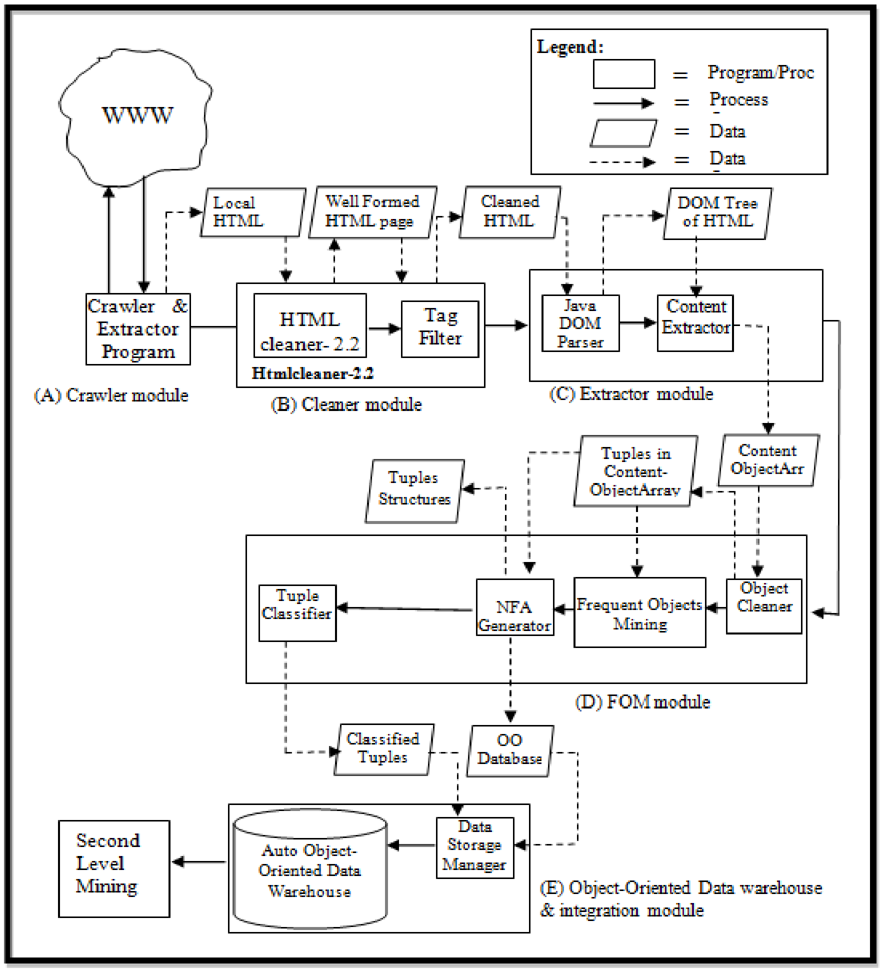


Figure 38: Advanced WebOMiner-3 Architecture.

```
Algorithm WebOMiner-3 Main
    Input:        Set of HTML files (WebPageFile) of web documents.
    Output:       Set of patterns of objects.
    Variable:     ContentObjectArray[].
Begin
      for each WebPageFile
          A.  Call Crawler Algorithm() to crawl webpage into local directory  from WWW. /* simple code to
                                  crawl the web page from the web and download it in local machine. */
          B.  Call HTMLCleaner-2.2 Software() to clean-up HTML code.        /* inside the code */
          C.  Call Extract() to create DOM tree of refined HTML file and extract web content objects
          sequentially from DOM Tree. Store objects in ContentObjectArray[].    /* figure 40 page 66 */

          D.  Call FrequentObjectsMining() to call NFA generator and identify data records patterns.
                                        /* figure 49 page 79 */
          F.  Call Create OODataWarehouse() to store data records into data warehouse /* figure 49 page 79 */
      end for
          G.  Mine for knowledge discovery within extracted contents. /* pending to develop*/
End
```

Figure 39: Advanced WebOMiner-3 main algorithm.

3.2.1 Crawler Module

We plan to use the crawler module which was proposed by Mutsuddy (2010). They proposed a mini-crawler algorithm to download the target web page and create a mirror of it, and then store it in the local computer.

3.2.2 Cleaner Module:

We use ‘HTMLCleaner-2.2’ software to clean the targeted web page. HTMLCleaner-2.2 is open source software and can be downloaded from http://htmlcleaner.sourceforge.net. Actually, DOM tree cannot be built properly if the given web page is not clean. The HTMLCleaner-2.2 is not enough to clean the web page, so we modified the cleaner module to clean the content of downloaded web page more properly. The cleaner now cleans all the noise contents such as ‘comment written by web developers’, ‘script’ codes, ‘flashes’, ‘meta data’.

3.2.3 Extractor Module

The extractor module takes the cleaned web page as input and creates the DOM tree logical structure for that page. We use Java DOM tree application package to build and traverse DOM tree for the given list product web page. Then the extraction process starts. Figure 40 shows the main extraction algorithm. Line 1.0 of the algorithm takes the cleaned web page as input and uses the Java DOM tree package to create the DOM tree of the given page. Line 2.0 calls the ContentObjectsExtraction() procedure to starts the extraction process.

```
Algorithm Extract()
Input:    Clean Web Page.
Output:  Populated ContentObjectArray[].
Begin
        1.1    Use Java DOM Tree Package to build DOM Tree.
        2.0    Call ContentObjectsExtraction() to extract data content objects /* figure 41
                                                                                    page 68 */
End
```

Figure 40: Extract() algorithm.

The new proposed extraction process is based on the idea of similarity between DOM tree tag nodes names. In the similarity matching the extraction process will be able to determine the boundary of each data region and all the data blocks inside a specific region. For each located data region a special text string ‘region’ will be inserted in ContentObjectArray[] array, and for each determined data block inside a specific data region an open-separator object ‘{‘ will be inserted in ContentObjectArray[] to refer for the beginning of the data block and close-separator object ‘}’ to refer for the end of the data block. Then extract and insert all the data objects which are related to a specific data block will be inserted between the open-separator object and the close-separator object. After the extraction process ends, ContentObjects-Extraction() procedure returns the ContentObjectArray[] which is filled with all the data regions that are separated on each other, and all data blocks that are related to a specific data region where each data block is surrounded by open-separator object ‘{‘ and close-separator object ‘}’. Figure 41 shows the pseudo code for ContentObjectsExtraction() procedure.

```
Procedure ContentObjectsExtraction()
Input:     DOM tree, website of page p
Output:  Populated ContentObjectArray[].
Begin
          1.0   while DOM tree has more nodes
              1.1    Node n= DOMtree.pareser.nextNode()
              1.2    if  n.getParsePath() != n.nextSibling().getParsePath()
                    1.3   if n.getParsePath() != n.previousSibling().getParsePath()
                       1.4   continue;
                    1.5   Else
                       1.6   ContentObjectarry[indexTag]='{' ;
                       1.7   indexTag++;
                       1.8   db= n.getParsePath();
                       1.9   if isProduct(db)
                              1.10  for each data object obj in db
                                1.11      objType=getObjectType(obj, website);
                                1.12      ContentObjectarry[indexTag]= objType;
                                1.13      indexTag++ ;
                              1.14  end for
                       1.15   Else
                              1.16  for each data object obj in db
                                1.17      objType=getObjectType(obj);
                                1.18      ContentObjectarry[indexTag]= objType;
                                1.19      indexTag++ ;
                              1.20  end for
                       1.21   ContentObjectarry[indexTag]='}' ;
                       1.22   indexTag++ ;
                       1.23   ContentObjectarry[indexTag]='region' ;
                       1.24   indexTag++;
                       1.25   for i=1 to n.getPathLength()
                                1.26      DOMtree.pareser.nextNode() ++ ;
                       1.27   end for
              1.28   Else
                       1.29   ContentObjectarry[indexTag]='{' ;
                       1.30   indexTag++ ;
                       1.31 db= n.getParsePath();
                    1.32 if isProduct(db)
                              1.33  for each data object obj in db
                                1.34      objType=getObjectType(obj, website);
                                1.35      ContentObjectarry[indexTag]= objType;
                                1.36      indexTag++ ;
                              1.37  end for
                    1.38 Else
                              1.39  for each data object obj in db
                                1.40      ContentObjectarry[indexTag]=obj ;
                                1.41      indexTag++ ;
                              1.42  end for
                       1.43   ContentObjectarry[indexTag]='}' ;
                       1.44   indexTag++ ;
                       1.45   for i=1 to n.getPathLength()
                                1.46      DOMtree.pareser.nextNode() ++ ;
                       1.47   end for
          2.0   End while
End
```

Figure 41: ContentObjectsExtraction() procedure.

Line 1.0 of the algorithm reads sequentially all the nodes of DOM tree and stops when there is no more nodes in DOM tree. Line 1.1 reads the next node of DOM tree and stores it in the ‘n’ Node variable. Line 1.2 does the similarity comparison between the pattern string path of the tag node ‘n’ and its next sibling. To get the next sibling for any node, DOM tree provides a function called ‘getNextSibling()’ returns the next sibling for the current node. The string pattern path of any specific node can be returned through the procedure called ‘getParsePath()’ which is proposed by us in this thesis and it will be described in subsequent lines. If there is no match between tag node ‘n’ and its next sibling, the comparison will be done between the current tag node ‘n’ and its previous sibling as per line 1.3. DOM tree provides a function called ‘getPreviousSibling()’ returns the previous sibling for the current tag node. If the match fails the algorithm continuous to the next node of DOM tree as per line 1.4, but if the match succeeds between the current tag node ‘n’ and its previous sibling, the algorithm inserts open-separator object ‘{‘ in the next position of ContentObjectArray[] as per line 1.6 because the algorithm considers the current tag node ‘n’ and all its sub-tree as a new data block. Then the algorithm extracts all data objects which are embedded underneath of the tag node ‘n’ and its sub-tree tag nodes as per lines 1.8-1.20. After it finishes, it inserts the close-separator object ‘}’ in the next available position of ContentObjectArray[] as per line 1.21. The algorithm considers the previous sibling of current node ‘n’ is the last sibling of the current tag node ‘n’ which represents the end of a specific data region, so it inserts the string ‘region’ at the next available position of ContentObjectArray[] as per line 1.23 to surround the data regions. Then the algorithm computes the length path of the tag node ‘n’ and all its sub-tree to move the pointer to the first node after last node in ‘n’s sub-tree as per lines 1.25-1.27. The length of tag node ‘n’ and its sub-tree will be computed through the function called ‘getPathLength()’ which is proposed by us in this thesis and will be described in subsequent lines. If the similarity comparison between the node ‘n’ and its next sibling matches as per line 1.2, then the else part of the algorithm will be executed as per line 1.28. It considers the current tag node ‘n’ and all its sub-tree as a new data block and inserts the open-separator object ‘{‘ in the next position of ContentObjectArray[] as per line 1.29. Then it extracts all data objects which are embedded underneath of the tag node ‘n’ and its sub-tree tag nodes as per lines 1.32-1.42. Then it inserts the close-separator object ‘}‘ in the next position of ContentObjectArray[] as per line 1.43. Then it computes the length of tag node ‘n’ and its sub-tree to move the current tag node to the first node after the last node in ’n’s sub-tree as per lines 1.45-1.47.

Example Extraction Process:

The following example illustrates the ‘Extract’ algorithm (figure 40) and ‘ContentObjectsExtraction()’ procedure (figure 41) and describes how they work. Let us suppose there is a snapshot of HTML code of web list product web page as shown in figure 42. For the example purposes we suppose this HTML snapshot is cleaned.

```
<html><head><title>Electronic Products Web Site</title></head>
<body><div>Electronic Products</div> <a href=’tv.html’> TVs </a> <a href=’computers.html’> Computers </a>
<a href=’monitors.html’> Monitors</a> <table> <tr><td  id=”title”>HP laptop core i5</td> <td>
<img src=”photos/phpi5.jpg” /> </td> <td id=”Brand”> HP </td><td id=”price”> $699 </td></tr><tr>
<td  id=”title”>Sony laptop core i7</td><td><img src=”photos/sonyi7.jpg” /></td><td id=”Brand”> Sony </td>
<td id=”price”>$899<td></tr> <tr><td  id=”title”>Dell laptop core i7</td> <td><img src=”photos/delli7.jpg” /></td>
<td id=”Brand”> Dell </td> <td id=”price”>$920</td></tr> </table> <a href=”tv1.html”><img src=”tv/im1.jpg /></a>
<a href=”tv2.html”><img src=”tv/im2.jpg /></a><a href=”tv3.html”><img src=”tv/im3.jpg /></a> </body> </html>
```

Figure 42: snapshot of HTML code of electronic products list web page.

The Extract() algorithm takes the HTML code of the web page shown in figure 42 as input, and then calls the Java DOM tree API package to create the DOM tree of that page as per line 1.0 of the algorithm. The generated DOM tree of the given electronic products list web page is shown in figure 43.

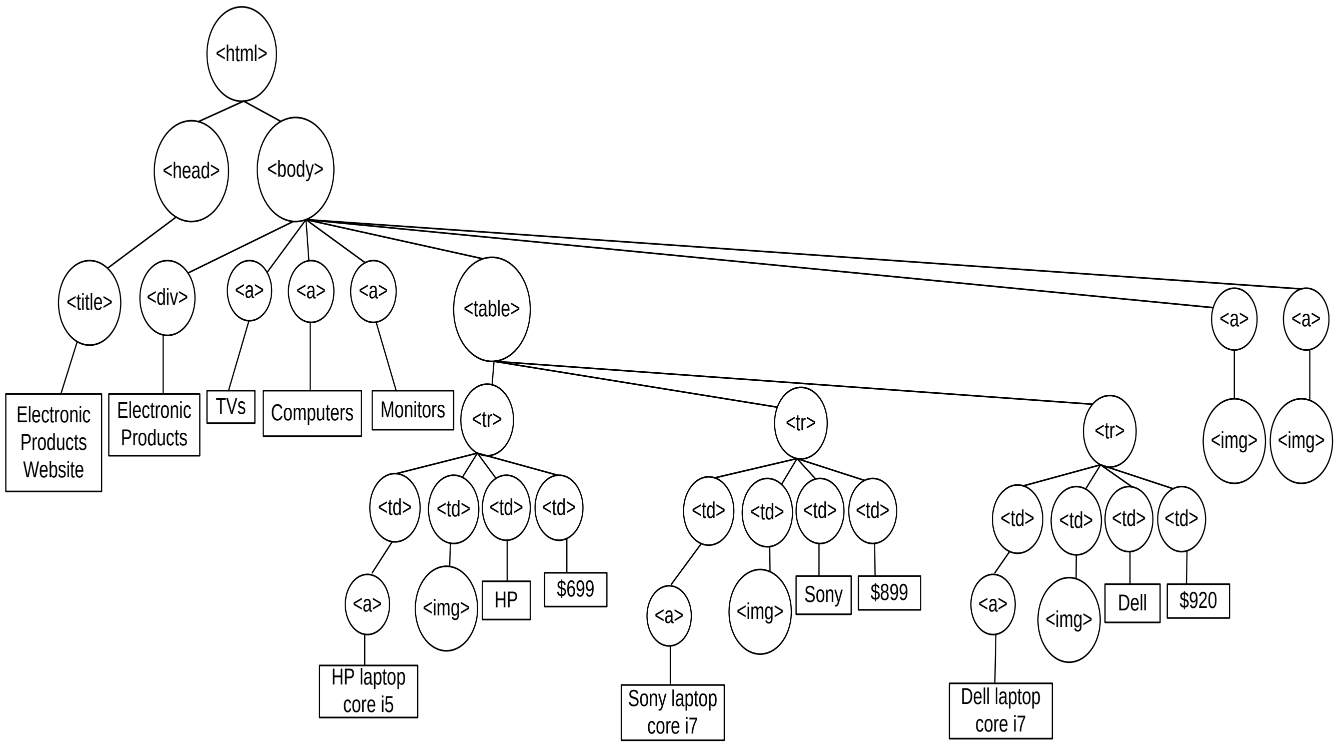


Figure 43: DOM tree of electronic products list web page.

After building the DOM tree, the Extract algorithm calls the contentObjectsExtraction() procedure as per line 2.0 (figure 41). The contentObjectsExtraction() takes the generated DOM tree (figure 43) as input and starts traverse the tree, and does the similarity between DOM tree tag nodes to extract the data objects and store them in ContentObjectArray[] array. The procedure initially assigns current tag node ‘n’ to the root of DOM tree which is the ‘<html>’ tag node as per line 1.1 (figure 41), the tag node ‘<html>’ has not any sibling so the procedure continuous as per line 1.4 and assigns the current tag node ‘n’ to the next tag node in DOM tree which is ‘<head>’. Since the tag node ‘<head>’ has only one sibling which is the tag node ‘<body>’, the procedure does the similarity comparison between the string pattern path of tag node ‘<head>’ and all its sub-tree and the string pattern path of tag node ‘<body>’ and all its sub-tree as per line 1.2. The string pattern of tag node ‘<head>’ and all its sub-tree is ‘*head title #text*’ which is not equal to the string pattern of tag node ‘<body>’ and all its sub-tree which is equal to ‘*body div #text a #text a #text a text table tr td a #text td img td #text td #text tr td a*

#text td img td #text td #text tr a td #text td img td #text td #text a img a img’ and since the tag node ‘<head>’ has no previous sibling, the procedure continuous and reads the next ‘n’ tag node which is ‘<title>’. As we mentioned previously that the function called ‘getParsePath()’ returns the string pattern of any given node and all its sub-tree. Since the tag node ‘<title>’ node has no siblings, the procedure reads the next tag node which is ‘<body>’ also it has no siblings, the procedure reads the next tag node ‘<div>’ and compares the string pattern path of tag node ‘<div>’ and all its sub-tree which equals to ‘*div #text*’ with the string pattern of tag node ‘<a>’ and all its sub-tree which equals ‘*a #text*’ since they are not similar the procedure continuous and reads the next node which is ‘<a>’. Since the tag node ‘<a>’ has two siblings, the procedure compares the string pattern of tag node ‘<a>’ which equals to ‘*a #text*’ with the string pattern of its next sibling ‘<a>’ tag node which equals to ‘*a #text*’, since the two string are the same the procedure knows this is the beginning of a data block node and the else part of the procedure at line 1.28 will be executed, so it inserts the open-separator object ‘{‘ in the first position of ContentObjectArray[] array as per line 1.29 (figure 41), and increments the counter indexTag as per line 1.30 to refer for the next available position of ContentObjectArray[] array. Once the procedure knows the beginning of the data block node, it assigns the currents data block to the variable db as per line 1.31 through calls the procedure getParsePath() of the current tag node which returns the complete structure patterns of data block. Then it checks if the current data block db is of type ‘product’ or not through calls the procedure isProduct() which determines if the db is product or not through our simple assumption to the current attributes order of product data block in currently being extracted list product web page of B2C website. The isProduct() procedure is deeply discussed in subsequent paragraphs. Since the current db ‘*a #text*’ is not of type ‘product’ because it does not meet the rule of product data block in current being extracted web page from ‘CompUSA’ website in this example, the procedure extracts all the data objects of data block node and all its sub-

tree and store them in ContentObjectArray[], so it inserts the '*link*' object in the second available position of ContentObjectArray[], and inserts the '*text*' in the third position as per lines 1.39-1.42. After extracts all the objects that are related to the current data block, the procedures know this is the end of the data block, so it inserts the close-separator object '}' in the next available position of ContentObjectArray[] array as per line 1.43. Then the procedure computes the length of current tag node 'n' which equals to '<a>' and all its sub-tree through the function called 'getPathLength()'. This function computes number of tag nodes elements in the sub-tree of the given node except the leaf nodes. Since the current tag node '<a>' has only one leaf node in its sub-tree the 'getPathLength()' will return zero and the procedure reads the next tag node '<a>' which is the second child of the '<body>' tag node as per lines 1.45-1.47. The same thing will be done for the second tag node '<a>', the procedure inserts the open-separator object '{', '*link*' object, '*text*' object, and close-separator object '}' in ContentObjectArray[] array. Then the procedure reads the next '<a>' node which is the third child of '<body>' tag node, since it has no next sibling the procedure compares it with its previous sibling as per line 1.3. The tag node '<a>' has string pattern equals to the string pattern of its previous sibling '<a>' node which equal to '*a #text*' so the similarity comparison succeeds and the procedure recognises a new data block node of non 'product' type, so it inserts open-separator object '{', '*link*' object, '*text*' object, and close-separator object '}' in the next available positions of ContentObjectArray[] array as per lines 1.16-1.20. Since the current tag node '<a>' is the last sibling, the procedures recognises that this is the end of a specific data region and inserts the string object '*region*' in ContentObjectArray[] as per line 1.23. The procedure continuous and reads the next node which equals to '<table>' tag node since it has no siblings, the procedure reads the next node which equals to '<tr>' tag node since it has the next sibling '<tr>' node, the procedure compares the string pattern of the current tag node '<tr>' which equals to '*tr td a #text td img td #text td #text'* with the string pattern of the next sibling tag node ‘<tr>’ which equals to ‘*tr td a #text td img td #text td #text’*, since the nodes are similar the procedure considers the current tag node ‘<tr>’ as a data block node and inserts the open-separator object ‘{‘ in ContentObjectArray[] as per line 1.29. Then the procedure assigns the current data block to the variable db as per line 1.31 through the procedure getParsePath(). Since the current db is ‘product’ data block because it meets the rules of product type data block by the procedure isProduct() as per line 1.32. Then the procedure extracts all the objects that are related to the tag node ‘<tr>’ and its sub-tree, and assign type of each ob_j through calls the procedure getObjectType() as per line 1.34. So the procedure inserts ‘*title=HP laptop core i5*’ object, ‘*img*’ object, ‘Brand=HP’ object, and ‘Price=$699’ object in the next available positions in ContentObjectArray[] as per lines 1.33-1.37. The procedures knows the end of the data block node and all its sub-tree, so it inserts the close-separator object at ContentObjectArray[]. The same thing will be done to the second ‘<tr>’ node, the following data objects: ‘link object, *title=Sony laptop core i7*’ object, ‘*img*’ object, ‘Brand=Sony’ object, ‘Price=$899’ object, and close-separator object ‘}’ in ContentObject-Array[] array. Also the same thing will done to the last ‘<tr>’ node through the comparison matching with its previous node the following objects: ‘*title=Dell laptop core i7*’ object, ‘*img*’ object, ‘Brand=Dell’ object, ‘Price=$920’ object, and close-separator object ‘}’ in ContentObject-Array[] array. Since the ‘<tr>’ node is the last sibling, the procedure recognises that this is the end of a specific data region, so it inserts the string object ‘*region*’ in ContentObjectArray[] as per line 1.23. The procedure continuous until reads all the tag nodes in DOM tree. Figure 44 shows the ContentObjectArray[] after finish the the contentObjectsExtraction() procedure and with this the Extract() algorithm will end. Actually, if we apply the WebOMiner and WebOMiner-2 to extract the data contents from the DOM-Tree shown in figure 43, both of the systems would fail to extract all the list and product data tuples in the page because they are looking for non-block level data blocks to start the extraction process, while all the lists and the products data blocks in the DOM-Tree are level data blocks, so the two system will not be able to identify the list and the product data blocks and they fail to extract them.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
{	link	text	}	{	link	text	}	{	link	text	}	region	{	link	title	image	brand	price	}	{	link	title	image	brand	price	}

27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
{	link	title	image	brand	price	}	Region	{	link	image	}	{	link	image	}

Figure 44: ContentObjectArray[] array after Extraxt() algorithm.

Figure 45 shows the pseudo code of getParsePath() procedure and will be described through application example 3.2.

```
Algorithm getPrasePath()
Input:    rootNode
Output:  the complete string path of rootNode and its sub-tree
Begin
        1.2    path = rootNode.getNodeName();
        1.3    node =rootNode.getFirstChild();
        1.4    while node!=null
             1.5    path =path || node. getNodeName();
             1.6    if node.hasChildNodes()
               1.7    if node.getNextSibling()!=null
                 1.8    stack.push(node.getNextSibling());
               1.9    node=node.getFirstChild();
             1.10  else
               1.11  node=node.getNextSibling();
               1.12  if node==null and !stack.isEmpty()
                 1.13  node=stack.pop();
        1.14  end while
        1.15  return path;
End
```

Figure 45: getParsePath() procedure.

Example Finding Path of Tag Node:

This example illustrates how the procedure getParsePath() (figure 45) works. Let suppose extracting the string pattern path of tag node ‘<tr>’ and its sub-tree shown in figure 46. The first line of getParsePath() procedure assigns the variable path with the name of rootNode which is ‘<tr>’ tag nodes, so the value of variable path will be ‘path=tr’. The built in procedure ‘getNodeName()’ returns the name of the current node. Then the procedure creates a new node ‘n’ and assigns it to the first child of the root node as per line 1.2, so the new node ‘n’ will be assigned to the first ‘<td>’ tag node in left side of figure 46. Then the procedure checks if the current ‘n’ node is not null node i.e., it checks if there is no more nodes in the path of root node as per line 1.3, otherwise the procedure will stop as per lines 1.13-1.14. The current ‘n’ tag node equals to ‘<td>’ and not equals to null, so the procedure concatenates the tag node ‘<td>’ name with the previous value of variable path as per line 1.4, so the new value of path will be ‘path=tr td’. Then the procedure checks if there are child nodes of the current node as per line 1.5. Since the ‘<td>’ node has one child node equals to ‘<a>’ tag node the procedure proceeds. The built in ‘hasChildNodes()’ procedure checks whether the current node has children nodes or not. Then the procedure checks if the current tag node ‘<td>’ has sibling node or not as per line 1.6. The build in ‘getNextSibling()’ returns the next sibling node of the current node. Since the ‘<td>’ tag has three siblings the procedure inserts the first sibling which is the second ‘<td>’ tag node in stack as per line 1.7. Then the procedure assigns the current ‘n’ node to the first child of ‘<td>’ tag node which equals ‘<a>’ tag node. The same process will be repeated for the current ‘<a>’ tag node. The path variable will be replaced by ‘path= tr td a’, since the tag node ‘<a>’ has one child and no siblings the current node will be the child node which equals to ‘#text’ so the path variable will be replaced as ‘path= tr td a #text’. Since the current node ‘#text’ has no child node the else part of the procedure will be executed as per line 1.9. Line 1.10 assigns the null value to the current node because ‘#text’ node has no siblings. Line 1.11 checks if the current node equals to null and there are no more nodes in the stack to end the procedure, but since the stack is not empty the procedure pops the top of the stack which contains the ‘<td>’ tag node and the while loop will be repeated again for the second ‘<td>’ tag node. The procedure continues traversing the whole sub-tree until no more nodes are available and the stack is empty. Finally it returns the complete path of the root node and all its sub-tree which equals to '*path = tr td a #text td img td #text td #text*' in the current example.

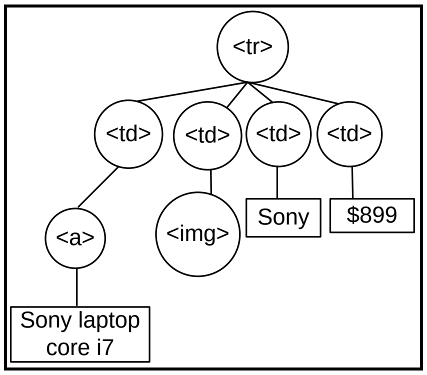


Figure 46: Root tag node and all its sub-tree.

Figure 47 shows the pseudo code of isProduct() procedure. It is a simple procedure takes the full pattern string of the current tag node and its sub-tree as input and returns Boolean 'true' value if the pattern is 'product' data block as per line 1.1-1.2 or 'false' if the pattern is not as per line 1.3.

```
Algorithm isProduct()
Input:    string pattern of tag node and its sub-tree
Output:  Boolean Flag ('true', 'false')
Begin
 1.1   if pattern.contains (at least one 'img' object and one
         'link' object and two '#text' objects)
      1.2 return 'true';
 1.3   else
      1.4 return 'false';
End
```

Figure 47: isProduct() procedure.

Figure 48 shows the pseudo code of getObjectType() procedure. This procedure takes the object order of 'product' data block and the website of web page which is being extracted as input and returns type of the object. Based on our simple observation of the product list web page in 'CompUSA' website, the order of 'product' data block as follows <link title img brand price>. The procedure checks if the website is 'compUSA' and the object index equals to 1 then it returns the 'link' as type of the object as per lines 1.1-1.3, if not the procedure checks if the object index equals to two then it returns 'img' as per lines 1.4-1.5, if not it checks if the object index equals to three then it returns 'brand' as per lines

1.6-1.7, if not it checks if the object index equals to four then it returns ‘price’ as object type as per lines 1.8-1.9, and the procedure ends.

```
Algorithm getObjectType()
Input:    object index, website
Output:  object type
Begin
  1.1    if website='compUSA'
      1.2 if object_index=1
        1.3    return 'link';
      1.4 else if object_index=2
        1.5    return 'img';
      1.6 else if object_index=3
        1.7    return 'brand';
      1.8 else if object_index=4
        1.9    return 'price';

End
```

Figure 48: getObjectType() procedure.

3.2.4 Frequent Objects Mining Module

As we discussed before that (Mutsuddy, 2010; Ezeife and Mutsuddy, 2013) proposed non full automatic approach to build NFA structure for each data tuple type such as product tuple, list tuple, text tuple, form tuple, singleton tuple, and noise tuple. They based on their observations to a limited number of web pages in domain of B2C websites to identify and build all the possibilities of the NFA structure for each tuple type. For example, in their work they suggested 10 different NFA’s structures for the product tuple to guarantee the identification of all products that might be in different formats and structures inside the observed list product web pages. This none full automatic approach is considered the main shortcoming of work by (Mutsuddy, 2010; Ezeife and Mutsuddy, 2013). In this module, we propose to build a full automatic approach to build NFA structure for each tuple type based on the already extracted data objects which are stored in ContentObjectArray[] array. Build automatic NFA will help to identify each data tuple inside ContentObjectArray[] array. Identifying the accurate type of any tuple helps us to know the right place of it in the data warehouse in order to store it correctly. In addition, automatic NFA would save a lot of time which is required by manual approach to build the same NFA structures. Also build NFA from current extracted data objects will suggest the exact required number of the required NFA structures to identify data tuples inside targeted web page, while in the manual approach the authors assume a fixed number of NFA types which equals to six in their work, and this limits the flexibility and extensibility of the proposed system, and it may miss identification of some important tuples as a result. Manual approach enforces the system to do an extra comparisons and processing steps which negatively effect on the performance of the system. Our proposed automatic approach is based on the proposed '*frequent objects mining*' (FOM) approach which aims to build regular expression of frequent objects patterns in ContentObjectArray[] array. Then build NFA structure of each regular expression. FOM builds the regular expression of frequent objects patterns through build frequent object growth tree (FO-growth Tree), and then traversing all the paths of the leaf nodes of FO-growth tree. Each generated path represents one regular expression. Figure 49 shows the proposed FOM algorithm.

```
Algorithm FrequentObjectsMining()
 Input: ContetObjectArray                    // Data structure contains content objects
 Output: A set of patterns of object's contents
     begin
         1.0  Call MineFrequentObjects()   /* figure 50 page 79 */
         2.0  Call GenerateNFA()            /* figure 58 page 86 */
     end;
```

Figure 49: Frequent Objects Mining (FOM) algorithm.

Line 1.0 of the FrequentObjectsMining() algorithm calls the MineFrequentObjects() procedure to build regular expression of the frequent objects r in ContentObjectArray[] array. Figure 50 shows the pseudo code of MineFrequentObjects() algorithm.

```
Procedure MineFrequentObjects()
Input: ContetObjectArray[]
Output: A set of frequent objects patterns
begin
     1.0  Call FO-Tree Construct()   /* figure 51 page 80 */
     2.0  Call generateRE()          /* figure 52 page 81 */
end;
```

Figure 50: MineFrequentObjects() algorithm

Line 1.0 of MineFrequentObjects() procedure calls the FO-Tree Construct() algorithm to build FO tree. This algorithm builds the logical data structure tree called '*frequent object tree*' (FO-tree) which represents the frequency of objects, and then traverses the FO-tree to finds the path of each leaf node in FO-tree. To clearly understand the FO-tree, we introduce here the basic definition of FO-tree:

- **Definition 2:** Frequent Object-tree (FO-tree) is a logical tree data structure with the following two main properties:

1. It consists of a root node with a label 'null' value, a set of object-prefix sub-trees as the children of the root.
2. Each node in the object-prefix sub-tree consists of three items: *object-name*, *count*, and *node-link*. Object-name identifies which object this node represents, count registers the number of tuples in ContentObjectArray[] array contains this object, and node-link links to the next node in FO-tree or to null if there is no next node.

Based on this definition, FO-Tree construction algorithm shown in figure 51.

```
Algorithm FO-tree Construct()
Input: ContetObjectArray[]
Output: FO-tree, the frequent object tree of ContetObjectArray[] tuples
begin
    1.0   Create the root of FO-tree, T, and assign it with 'null' value.
    2.0   for each data block $db_i$ in ContetObjectArray[]
      3.0   for each $ob_j$ in $db_i$
              4.0   if T has a child node such that n.object-name  = $ob_j$.object-name
                5.0    n.count++
              6.0   else
                7.0    create a new node n in T
                8.0    n.count=1
                9.0    n.name= $ob_j$.object-name
                10.0   n.parent node-link refer to $ob_j$
      11.0  end for
    12.0  end for
    13.0  return T
end
```

Figure 51: FO-tree Construct() algorithm.

Line 1.0 of FO-tree Construct() algorithm creates the root node of FO-tree, T, and assigns it with the 'null' value. Then for each data block db_i in ContentObjectArray[] which represents the sequence of one or more objects ob_j, the algorithm takes each object ob_j and searches for the node '*n*' in '*T*' which has the same object name value of ob_j as per line 4.0 (figure 51). If it finds the node '*n*', the algorithm increments the node *n.count* by 1 as per line 5.0. If it does not find the node '*n*', the algorithm creates a new node '*n*' and sets 'n.count=1' and assigns the '*n.object-name*' by 'ob_j.*object-name*', and links the new '*n*' node with the '*T*' via '*n.parent-link*' as per lines 7.0-10.0.

Line 2.0 of MineFrequentObject() algorithm (figure 50) calls the generateRE() algorithm which generates the regular expression of each leaf node's path in FO-tree. Figure 52 shows the pseudo code of generateRE() algorithm.

```
Algorithm generateRE()
  Input: FO-tree
  Output: the complete set of regular expressions
  Begin
  1.0   generateLeafNodesPaths();
  2.0   for each leaf node path in FO-tree
        2.1 if freq(path) > =1 then
              2.2 regEx='(';
              2.3 for each node in path
              2.4 regEx= regEx || node.GetNodeName();
              2.5 end for
              2.6 regEx= regEx || ')' || '+';
        2.7 else
              2.8 regEx=null;
  3.0   end for
  End
```

Figure 52: generateRE() algorithm.

Line 1.0 of generateRE() algorithm generates all the paths to leaf nodes of FO-tree through calling the procedure generateLeafNodesPath(). Line 2.0 takes each path and generates the regular expression of that path. Line 2.1 checks the frequency of the path if it is greater than or equal to one, if so the algorithm assigns the open bracket '(' symbol to regEX variable as per line 2.2. Then the algorithm concatenates the name of each node in the path in consequence order as per lines 2.3-2.5. Line 2.6 assigns the close bracket ‘)‘ symbol to regEX variable referring to the end of the regular expression, and assigns the ‘+’ symbol to the end of the regEX referring that this regular expression has repeated one or more times as per line 2.6.

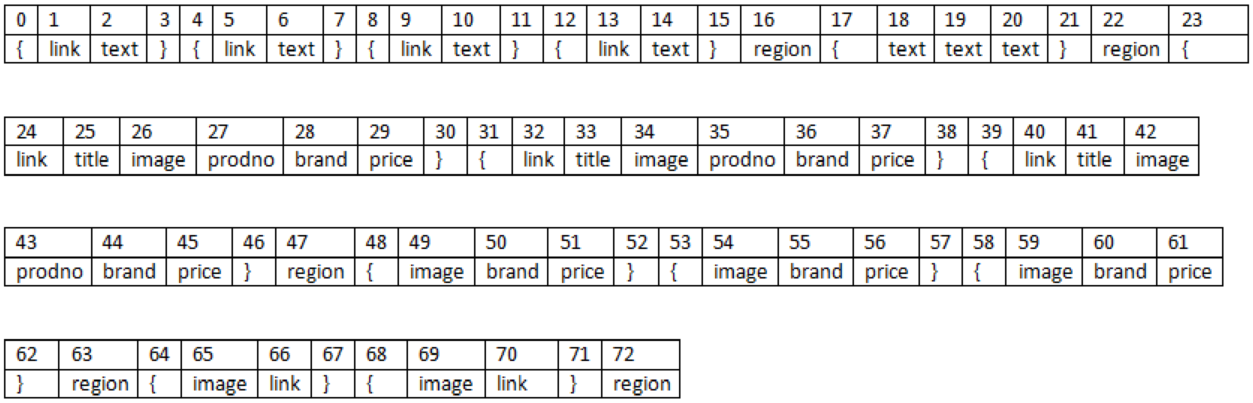
Figure 53 shows the pseudo code of generateLeafNodesPaths() procdure. Line 1 of the procedure initializes the variable pathlen with zero value and line 2 initializes the path[] array with null value. Line 3 checks if the current node is not, if so the node’s name will be inserted in the current available position of path[] array as per line 3.1, and the pathlen counter will be incremented as per line 3.2. Then the procedure checks if the current node is a leaf node or not, if it is a leaf node it inserts the complete path of this leaf node starting from the root node into the setPaths[] array. But if the current node is not a leaf node then the procedure is called recursively again two times for both the left node of current node and for the right node of the current node.

```
Algorithm generateLeafNodesPaths ()
  Input: FO-tree, path [], pathlen
  Output: setPaths[] /*the complete set of paths */
  Begin
  1.0   pathlen=0
  2.0   path[]=null
  3.0   if node is not null then
        3.1    path[pathlen]=node.GetNodeName();
        3.2    pathlen++
        3.3    if node is a leaf node then
          3.4    path[pathlen]=node.GetNodeFrequency();
          3.5    add path to setPaths[]
        3.6    else
          3.7 generateLeafNodesPaths(node→left, path, pathlen)
          3.8 generateLeafNodesPaths(node→right, path, pathlen)

  End
```

Figure 53: generateLeafNodesPaths() procedure.

Example Generating Leaf Node Path:

This example illustrates the MineFrequentObjects() procedure (figure 50) and how it works. It calls FO-tree construct() algorithm (figure 51) which builds frequent objects tree (FO-tree), and calls generateRE() algorithm (figure 52) which traverses the built FO-tree and generates the regular expressions of frequent object patterns. Let the MineFrequentObjects() procedure takes the ContentObjectArray[] which contains 72 data objects shown in figure 54 as input.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
{	link	text	}	{	link	text	}	{	link	text	}	{	link	text	}	region	{	text	text	text	}	region	{

24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
link	title	image	prodno	brand	price	}	{	link	title	image	prodno	brand	price	}	{	link	title	image

43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61
prodno	brand	price	}	region	{	image	brand	price	}	{	image	brand	price	}	{	image	brand	price

62	63	64	65	66	67	68	69	70	71	72
}	region	{	image	link	}	{	image	link	}	region

Figure 54: ContentObjectArray[] array after extraction from DOM Tree.

Line 1.0 of procedure calls the FO-tree Construct() algorithm (figure 51) which builds the frequent object tree as shown in figure 55. Each node in FO-tree represents the object name and repetition of the object with its sequence except the root of FO-tree which is labelled with '*null*' value. For example in FO-tree shown in figure 55, the left node of the first level of FO-tree contains the object '*link:7*' and this means that the object 'link' has repeated seven times. Line 2.0 of MineFrequentObjects() procedure calls the generateRE() algorithm (figure 52) to generate regular expressions. generateRE() takes the generated FO-tree in figure 55 as input and generates all the paths of each leaf node in FO-tree. Each path consists of a sequence of one or more nodes names and the length of this path, where the length of path can be represented as the frequency of the leaf node. For example, the path of the 'text' leaf node at the left most of FO-tree is 'link text (4)', where 4 represent the frequency of this path and can be found in the leaf node.

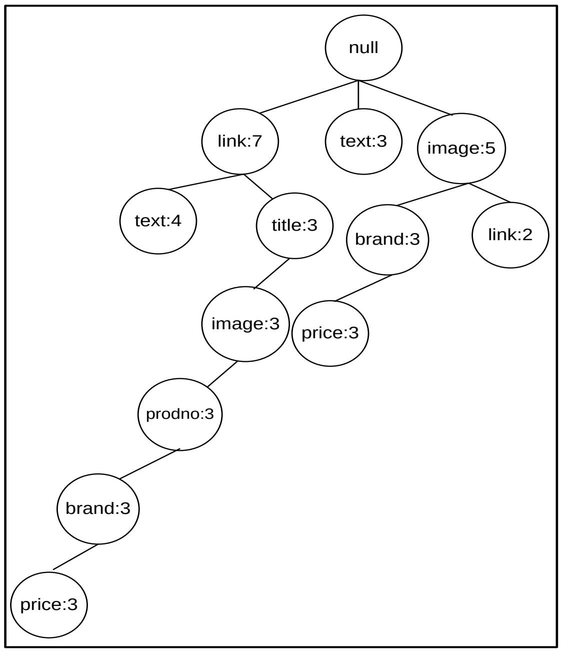
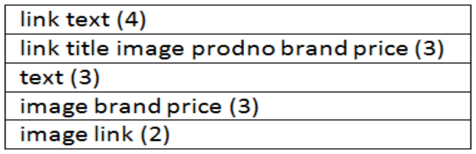


Figure 55: FO tree.

Line 1.0 of generateRE algorithm calls generateLeafNodesPaths() procedure (figure 53) to find all paths of leaf nodes of FO-tree. Lines 1.0-2.0 of the procedure initializes the variables pathlen and path[]. Line 3.0 checks if the current node is not null, if it is not the procedure inserts the name of the node in the current position of path[] array and increments the pathlen counter as per lines 3.1-3.2. Line 3.3 checks if the current node is leaf node or not. So if the current node is a leaf node then the procedure inserts the frequency of the node in the last position of path[] array as per line 3.4, and adds the current generated path to the setPaths[] as per line 3.5. But if the current node is not leaf node then the procedure is called two time recursively one for the left node of the current node and the other one for the right node of the current node. Figure 56 shows all the paths of leaf nodes of FO-tree in this running example.



link text (4)
link title image prodno brand price (3)
text (3)
image brand price (3)
image link (2)

Figure 56: Paths of leaf nodes.

Line 2.0 of generateRE() algorithm takes each generated path and build the regular expression of that path. Figure 57 shows all the regular expressions of the generated paths.

Path	Regular Expression
link text (4)	$(\text{link text})^+$
link title image prodno brand price (3)	$(\text{link title image prodno brand price})^+$
text (3)	$(\text{text})^+$
image brand price (3)	$(\text{image brand price})^+$
image link (2)	$(\text{image link})^+$

Figure 57: Regular expressions of generated paths.

Line 2.0 of FOM algorithm (figure 49) calls GenerateNFA() algorithm to generate the NFA structure for each data tuple type which will be used to identify tuples in ContentObjectArray[] array. Figure 58 shows the pseudo code of GenerateNFA() algorithm. The algorithm sequentially scans each regular expression generated by generateRE() algorithm and starts automatically builds the NFA structure of each regular expression. For example, the GenerateNFA() algorithm picks the regular expression '$(\textit{link text})^+$' since the initial NFA is not existed, the algorithm creates the initial state 'q_0' and refer to it as the current state 'g_c' to 'q_0' as per line 2.0. For '*link*' object, it creates a new state 'q_1' and refers to it as next state by the header of 'q_0' state. Store '*link*' object as the transition between state 'q_0' and state 'q_1' as per line 3.3.1. For the next iteration it scan the '*text*' object and creates a new state 'q_2', and refers to it by 'q_1', store '*text*' object as the transition between state 'q_1' and state 'q_2' as per line 3.3.1. Since '*text*' is the last object, it is denoted as the last state 'F' as per line 3.4. Figure 59 shows the NFA structure of the text-list data tuple.

```
Algorithm GenerateNFA (Enum x)
Input:      Enumeration x                          //Pattern Table of specific tuple type x
Output:     Seed NFA of tuple type x
Begin
If  NFA exist
  1.1  set qc ← q0;
    2.0 else
          2.1  initialize data structure for NFA, N= (Q, Σ, δ, q0, F);
          2.2  set Q ← {q0}, δ ← 0, F ← 0;
          2.3  set qc ← q0;
    3.0 for each object ‘o’ in Regular_Expression
          3.1  If  ∃ δ(qc , o) = qn or ∃ δ(qc , ε) = qj ∧ δ(qj , o) = qn in Seed NFA
                3.1.1  set qc ← qn;
          3.2  Else if ∃ δ(qc , o´) = qn ; where o´ ≠ o          //create ε transition
                3.2.1 create new state qa ; a < c
                3.2.2 create transition δ(qa , ε´) = qc ;         // i.e., δ ← δ U{((qa , ε´), qc)}
                3.2.3 set qc ← qa , Q ← Q U {qc} ;
                3.2.4 create transition δ(qc , ε´) = qj , qc ← qj ;   here c < j
                3.2.5 create new state qm and δ(qc , o) = qm ;     // i.e., δ ← δ U{((qc , o), qm)}
                3.2.6 set qc ← qm , Q ← Q U {qm} ;
          3.3 else
                3.3.1 Create new state qc+1 and δ(qc , o) = qc+1 ;   // i.e., δ ← δ U{((qc , o), qc+1)}
                3.3.2 set qc ← qc+1 , Q ← Q U {qc+1} ;
             Endif
          3.4 If ‘o’ is the last object in RE
                3.4.1 If Q ∩ qc+1 = 0;
                          Set qc ← F;
                3.4.2 Else
                          Refine Seed NFA to create representation pattern;
                       Endif
            endif
      endfor
End
```

Figure 58: GenerateNFA() algorithm.

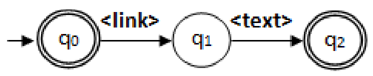


Figure 59: NFA structure of text-list tuple.

The algorithm continues building other tuples NFA structures. For example there are two product tuples in regular expression list shown in figure 56. In this case the NFA generator merges the two NFA’s structure in one NFA structure for product tuple. The first regular expression is ‘(link *title image prodno brand price)*$^+$’ and the second regular expression is ‘(*image brand price)*$^+$’, so NFA merges them to one NFA structure for product tuple as shown in figure 60.

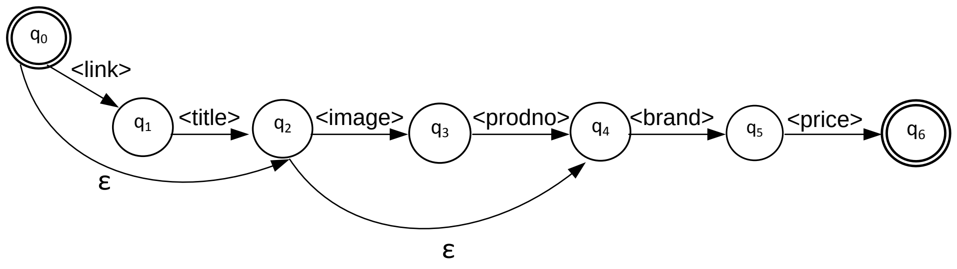


Figure 60: NFA structure of product tuple.

The NFA structure of image list tuple is shown in figure 61 and the NFA structure of text tuple is shown in figure 62.

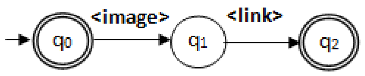


Figure 61: NFA structure of image-list tuple.

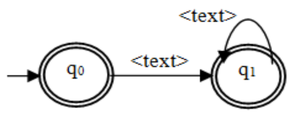


Figure 62: NFA structure of text tuple.

After identifying the structure of product data tuple, the algorithm extracts the specification attributes for each product tuple in list product web page from its own detailed web page.

3.3 OO Data Warehouse and Integration Module

The main purpose of the proposed Object-Oriented Data Warehouse (OODWH) is to integrate and store the extracted data tuples from different B2C websites in hierarchal and historical order for further processing and knowledge extraction. In addition, the proposed OODWH serves as a data repository for information about different types of products which are browsed online where the end user can query such data and get all the answers to do a comparison shopping between these products. The WebOMiner-3 generates the complete website schema after calling the NFA generator algorithm which generates the NFA structures of all data block types such as product, list, and text tuples. After generating the website schema, the WebOminer-3 automatically creates the OODB for that website separately based on generated schema and then fetches the data contents from the ContentObjectArray[] array and stores them in the right place in OODB tables. Once a new website is extracted and built in OODB, the WebOminer-3 starts the integration process. In the integration process, the WebOMiner-3 matches all the columns of all super-classes of all OODB sources and build a unique superclass, then does the same columns matching of sub-classes and creates a new sub-classes tables inherit the main superclass table. Each new subclass table will be a subclass and dimension table at the same time. Also the WebOminer-3 does the matching between ‘List’, ‘Text’, ‘Form’, ‘Noise’ tables for all OODB sources and creates a new dimension tables named as the following: ‘List_Dim’, ‘Text_Dim’, ‘Form_Dim’, ‘Noise_Dim’. Then the WebOMiner-3 creates two fact tables to integrate data contents from different OODB sources. The first fact table is called ‘ComputersFact’ which stores information about all computers ‘Desktops’ and ‘Laptops’ and their prices, OODB source, and their historical information. The second fact table is called ‘TuplesFact’ which stores historical information about all tuples of type ‘List’, ‘Text’, ‘Form’, ‘Noise’ from all OODB sources. The data warehouse will be in star schema, where each fact table represents the center table and all other dimension tables are directly connected to it through the primary and foreign key relationships. Figure 63 shows the pseudo code of CreateOODWH() algorithm. Line 1.0 of the algorithm creates the connection to Oracle database server. Line 2.0 checks if the current website OODB schema is already existed or not. If the schema is existed, the WebOMiner-3 checks for the new updates in the schema and does the new modifications as per lines 2.1-2.2. If the schema is not existed, that means that the website is a new source, then the WebOMiner-3 creates the OODB separately based on the given website schema which was also generated by WebOMiner-3 as per line

3.1. Then the WebOMiner-3 checks if the number of OODB sources is greater than one, if so the WebOMiner-3 starts the integration process between all the OODB sources.

```
Algorithm CreateOODWH()
Input:  Website schema
Output:   Populated data warehouse
Begin:
  1.0  Register Oracle driver and create Oracle connection.
  2.0  if the current website schema is already existed
            2.1  checkUpadate();
            2.2  if there is any new modification
                      2.2.1 modify();
  3.0  else
            3.1  create OODB(website schema);
            3.2  if different OODB >1
            3.3  for all super-classes in all OODB sources
                      3.3.1 matching();
                      3.3.2 create unique superclass();
            3.4  end for
            3.5  for all sup-classes in all OODB sources
                      3.5.1 matching();
                      3.5.2 create all unique sub-classes ();
                      3.5.3 each new subclass inherits superclass;
                      3.5.4 rename new subclass to newname_dim;
            3.6  end for
            3.7  for all tuple tables in all OODB sources
                      3.7.1 matching();
                      3.7.2 create all unique tuple tables ();
                      3.7.3 rename each new tuple table to newname_dim;
            3.8  end for
            3.9  create fact table 'ComputersFact' ;
            3.10  connect all sub-classes dimension tables with
                  'ComputersFact' fact table;
            3.11 create fact table 'TuplesFact';
            3.12  connect all tuples dimension tables to 'TuplesFact' fact table
  4.0  populate Data();
  5.0  end if
End
```

Figure 63: CreateOODWH() algorithm.

Line 3.3 of the algorithm (figure 63) does the matching between all super-classes that are located in all OODB sources, and creates one unique superclass contains all the matched or similar attributes as per lines 3.3.1 and 3.3.2. Then the algorithm does all the matching between all similar sub-classes attributes in all OODB sources and creates a unique subclass for each similar sub-classes that share the similar attributes as per lines 3.5.1 and 3.5.2. Then the algorithm creates the hierarchal relationship between the sub-classes and the superclass, where each subclass inherits the superclass as per line 3.5.3. Then the algorithm names each subclass table by the newname_dim and that means that each subclass table will be a dimension table too as per line 3.5.4. Then the algorithm does the matching between all other tuples attributes for all OODB sources and creates a unique table for similar tables that share the similar attributes as per lines 3.7.1 and 3.7.2. Then the algorithm names each tuple table by newname_dim because each tuple table will be a dimension table as per line 3.7.3. Then the algorithm create the main fact table called 'ComputersFact' which will store the integrated and historical information about all types of computers (desktops, and laptops) between all OODB sources as per line 3.9, then the algorithm connects all dimension sub-classes (tables) to the 'ComputersFact' table through the primary key and foreign key relationships as per line 3.10. Then the algorithm creates the second fact table called 'TuplesFact' which stores integrated and historical information about all other tuples types except product tuples from all OODB sources as per line 3.11. Then the algorithm connects all non-sub-class dimension tables to the 'TuplesFact' through the primary key and foreign key relationships as per line 3.12. Finally, the WebOMiner populates the data from all OODB sources into the new integrated OODWH as per line 4.0.

Example Build OODWH Schema:

Let us suppose that the generated OODB Computer schema of 'compUSA' website as the following:

Computer as superclass table inherits the root superclass=null has the structure:

Computer = <Computer_ID, Computer_type, title, brand, price, memory_size, memory_type, processor_type, processor_speed, OS >.

Laptop as subclass inherits the Computer superclass has the structure:

Laptop = < Laptop_ID, Computer, screen_size, touch_screen, weight, color, battry_type, integrated_webcam >

Desktop as subclass inherits the Computer superclass has the structure:

Desktop = < Desktop_ID, Computer, form_factor, Bays, line_inJack, PS2_connector >

List tuples table has the structure**:**

List = < List_ID, link, text >

Text tuples table has the structure**:**

Text = < Text_ID, Text >

Form tuples table has the structure**:**

Form = < FormID, Type, context >

Noise tuples table has the structure**:**

Noise = < NoiseID, Type, context >

The CreateOODWH() algorithm (figure 63) checks if the OODB schema of 'compUSA' website is existed or not in OODWH, it finds that the OODB schema of 'compUSA' is new so the algorithm creates the OODB schema separately as shown in figure 64.

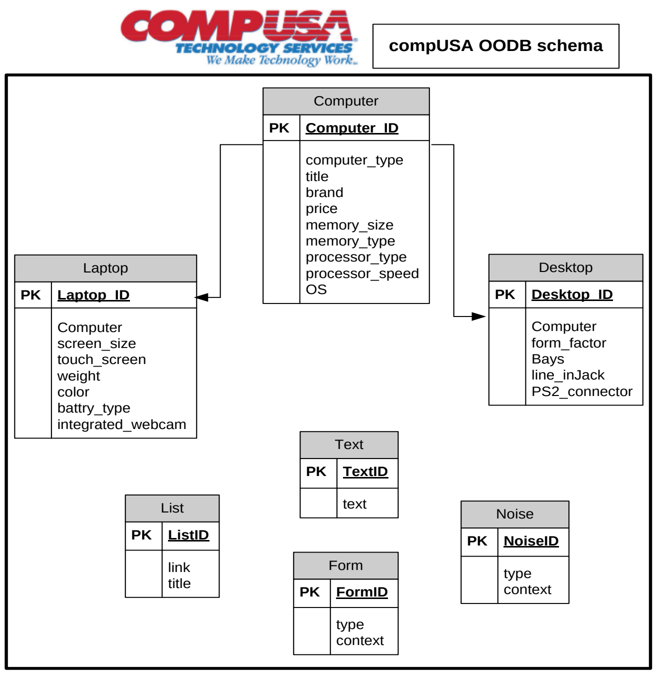


Figure 64: CompUSA OODB schema.

Let us suppose that the WebOMiner-3 extracts a new OODB schema for ‘BestBuy’ website, the generated schema is defined as the following:

Computer as superclass table inherits the root superclass=null has the structure:

Computer = <Computer_ID, Computer_type, price, title, brand, RAM_size, RAM_type, processor_class, processor_speed, OS >.

Laptop as subclass inherits the Computer superclass has the structure:

Laptop = < Laptop_ID, Computer, screen_size, touchScreen, weight, color, battry_type, webcam >

Desktop as subclass inherits the Computer superclass has the structure:

Desktop = < Desktop_ID, Computer, form_factor, Bays, lineJack, PS2 >

List tuples table has the structure**:**

List = < List_ID, link, text >

Text tuples table has the structure**:**

Text = < Text_ID, Text >

Form tuples table has the structure**:**

Form = < FormID, Type, context >

Noise tuples table has the structure**:**

Noise = < NoiseID, Type, context >

The CreateOODWH() algorithm (figure 63) checks if the OODB schema of ‘BestBuy’ website is existed or not in OODWH, it finds that the OODB schema of ‘BestBuy’ is new so the algorithm creates the OODB schema separately as shown in figure 65.

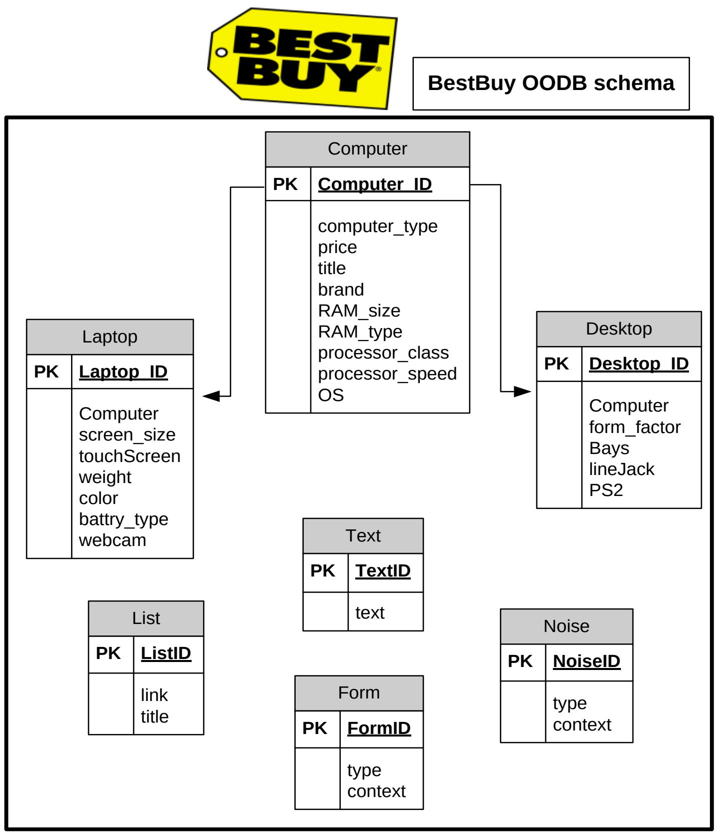


Figure 65: BestBuy OODB schema.

The CreateOODWH() algorithm figures out that the number of OODB schemas in OODWH is greater than one, so the algorithm starts the integration process between all the OODB sources. Initially, the algorithm compares the superclass tables in all OODB schemas sources and finds the matched or similar attributes, so the algorithm creates one superclass table called 'Computer' has the similar attributes between 'Computer' superclass table in 'compUSA' OODB and 'Computer' superclass table in 'BestBuy' OODB. The new 'Computer' superclass table in OODWH has the following structure after the matching process is done:

Computer as superclass table inherits the root superclass=null has the structure:

Computer = <ComputerID, ComputerType, title, memorySize, memoryType, processorClass, processorSpeed, OS >.

Then the CreateOODWH() algorithm compares the similar subclass tables in all OODB schemas sources and finds the matched or similar attributes, it compares the ‘Laptop’ subclass tables structures between OODB schema of ‘compUSA’ and OODB schema of ‘BestBuy’. The new ‘Laptop’ subclass table in OODWH has the following structure after the matching process is done:

Laptop = < LaptopID, Computer, screenSize, touchScreen, weight, color, battryType, webcam >

Then the algorithm renames the ‘Laptop’ subclass table to ‘Laptop_Dim’ and creates the hierarchal relationship between ‘Laptop_Dim’ subclass and ‘Computer’ superclass, where ‘Laptop_Dim’ inherits ‘Computer’.

Then it compares the ‘Desktop’ subclass tables structures between OODB schema of ‘compUSA’ and OODB schema of ‘BestBuy’. The new ‘Desktop’ subclass table in OODWH has the following structure after the matching process is done:

Desktop = < DesktopID, Computer, formFactor, Bays, lineJack, PS2 >

Then the algorithm renames the ‘Desktop’ subclass table to ‘Desktop_Dim’ and creates the hierarchal relationship between ‘Desktop_Dim’ subclass and ‘Computer’ superclass, where ‘Desktop_Dim’ inherits ‘Computer’.

Then the algorithm creates the main fact table called ‘ComputersFact’ which integrates the historical information about all computer products (desktops, laptops) from all OODB sources. The ‘ComputersFact’ table has the following structure:

ComputersFact = < ComputerID, brand, price, website, extractedDateTime >

Then the algorithm connects the two new subclass tables ‘Desktop_Dim’ and ‘Laptop_Dim’ to the main fact table ‘ComputersFact’ in a star schema through the primary and foreign keys relationships.

Then the algorithm compares the similar tuples tables between all OODB sources and creates a unique tuple table for each tuple type and makes it as a dimension table, so the new dimension tuples tables will be as the following:

List_Dim = < List_ID, link, text >

Text_Dim tuples table has the structure**:**

Text_Dim = < Text_ID, Text >

Form_Dim tuples table has the structure**:**

Form_Dim = < FormID, Type, context >

Noise_Dim tuples table has the structure**:**

Noise_Dim = < NoiseID, Type, context >

Then the algorithm creates the second fact table called ‘TuplesFact’ which integrates the historical information about all tuples except computers tuples from all OODB sources. The ‘TuplesFact’ has the following structure:

TuplesFact = < TupleID, type, website, extractedDateTime >

Then the algorithm connects all the tuples dimension tables, except subclass dimension tables of computers products, to fact table ‘TuplesFact’ in a star schema through the primary and foreign keys relationships. The final OODWH schema will be as shown in figure 66. After building the complete structure of OODWH, the CreateOODWH() algorithm populate the data from all the OODB sources and loads it in the right tables in OODWH.

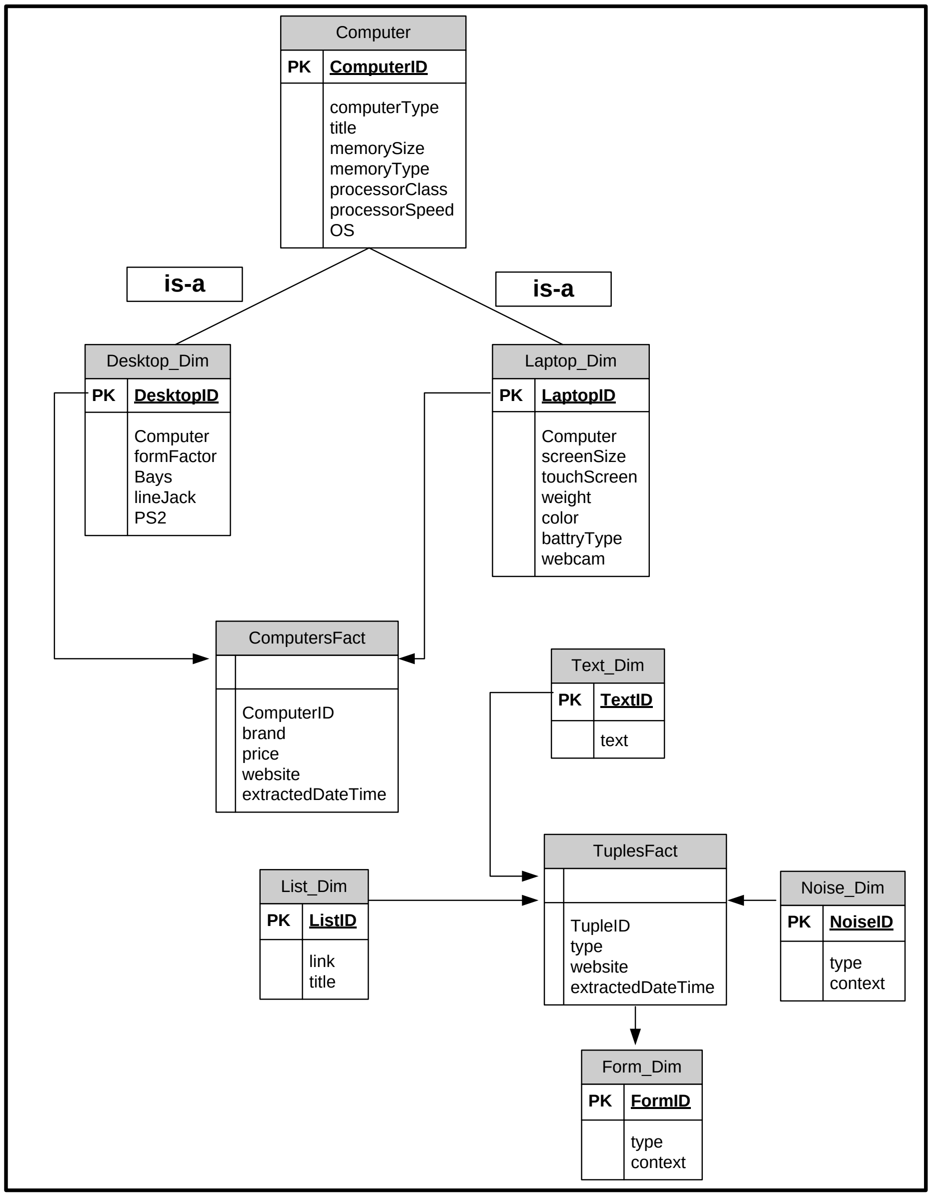


Figure 66: diagram for the initial OO 'Computer' object data warehouse.

3.4 Complexity Analysis

The complexity analysis of our system WebOMiner-3 focuses on the sequential execution of major processes in extraction, frequent objects mining, and object-oriented data warehouse modules and neglects the crawler and cleaner modules due to the lightweight processes are done there. The worst case execution time of the extraction algorithm is O(N^2), where N represents the total number of tag nodes in DOM-tree. In the worst case the algorithm needs to compare each node n with all the other nodes in DOM-Tree. The worst case of the frequent object mining module is O(MN), where M is the number of distinct data tuples that can be generated and the N is the cost of building the NFA for each distinct data tuple type. The worst case execution time of OODWH module is O(MN^2), where M is the number of websites needs to be integrated into the OODWH and N represents the total number of distinct tables will be built in the data warehouse. Also this process includes matching the common and distinct attributes of tables to build the object-oriented data warehouse. As a result, the total execution time of WebOMiner-3 system in the worst case will be O(N^2)+ O(MN)+ O(MN^2) which can be summaries as O(N^2).

CHAPTER- 4 Evaluation of WebOminer-3

We are done with the implementation of WebOminer-3 and working to enhance the proposed algorithms to get more accurate and scalable results. WebOMiner-3 is an extension of WebOMiner and WebOMiner-2 which are unique works for mining web contents in object-oriented approaches. A comparison performance results between these three systems are given in this chapter, but a valid comparison with other extraction and mining systems is not existed in this thesis.

4.1 Strength of WebOMiner-3

WebOMiner-3 is automatic system to extract web contents from web based on object-oriented model. Many of extraction and mining system have been proposed in the literature. Earlier systems use supervised, unsupervised, and wrapper generation techniques to extract contents from the web. IEPAD is a semi-unsupervised extraction system recognises repeated patterns by building tag-tree and uses center star method to extract targeted web contents. RoadRunner is unsupervised extraction system which generates the wrapper as regular expression from set of matched web pages by matching HTML tokens and collapses the unmatched HTML tokens. These systems are different from automatic WebOMiner-3 system due to the wide variance in the extraction process. DEPTA is semi-supervised extraction system uses DOM-Tree logical structure to analyse the web page and facilitates the extraction process of its contents. DEPTA uses one web page as training page to mark the targeted web contents and then builds the wrapper to extract similar web contents from other web pages. DEPTA is closer to WebOMiner-3 in terms of presenting web page as DOM-Tree, so we compare our system with DEPTA beside the comparison with WebOMiner and WebOMiner-2. The comparative analysis between WebOMiner-3 and DEPTA is summarised in table 2, between WebOMiner-3 and WebOMiner is summarised in table 3, and between WebOMiner-3 and WebOMiner-2 is summarised in table 4.

DEPTA	WebOMiner-3
Does not focus on the correctness of HTML code	Clean the HTML code very well that the DOM-Tree is built correctly.
Uses the web browser to get visual coordination of HTML tags hierarchical positions to build DOM-Tree and then traverses the tree to extract web contents.	Uses the DOM-Tree java package to build the tree which does not depend on the location of tag nodes to build the tree.
It extracts only textual contents type from the web.	It extract different types of web contents based on the tag nodes and their attributes such as Text, Image, Links, etc.
Does not provide any implementation for the database.	Automatically builds the OODWH and stores the extracted data in the right tables of the data warehouse.
It extracts only the product data objects.	It extracts different types of data objects such as product, list, text, form, noise.

Table 2: Comparative analysis between DEPTA and WebOMiner-3.

WebOMiner	WebOMiner-3
It builds the NFA structure of product data block based on the manual observations of the authors.	It builds the NFA structures of all data block types automatically based on the extracted data contents.
It extracts the schema for only one website.	It generates the schema for more than one website.
It does not integrate the extracted data contents from different websites.	It integrates the extracted data contents about different data tuples (product, list, text) from different websites using the data warehouse.
It uses only the htmlcleaner-2.2 to clean the web page before build the DOM-tree.	It uses the htmlcleaner-2.2 and new cleaner procedure to clean the comments, scripts, style sheets, and metadata inside the webpage.
It uses the concepts of level and non-level data blocks definitions to identify data blocks during traversing the DOM-Tree which is a very costly process that requires checking every tag node in DOM-Tree.	It proposes a new definition for both data block and data region and proposes a new efficient similarity matching technique of DOM-Tree tag nodes to locate the data blocks and data regions.
It stores the extracted data contents in relational database (RDB) which is neither historical nor object-oriented.	It stores the extracted data contents in object-oriented data warehouse (OODWH) which is more appropriate and beneficial due to the nature of the extracted data contents.

Table 3: Comparative analysis between WebOMiner and WebOMiner-3.

WebOMiner-2	WebOMiner-3
It extracts the schema for only one website.	It generates the schema for more than one website.
It automatically generates the NFA structure of different data block types based on the frequency of data attributes and generating the regular expression of the frequent patterns. It does not take into the consideration the sequence order of data attributes, so it generates many of extra candidate patterns of data blocks.	It automatically generates the NFA structure of different data block types based on the frequency and the sequence order of data attributes and generates a specific number of candidate patterns of data blocks.
It does not integrate the extracted data contents from different websites.	It integrates the extracted data contents about different data tuples (product, list, text) from different websites using the data warehouse.
It extracts data contents from only list product web pages.	It extracts data contents from both list and detailed product web pages.
It uses only the htmlcleaner-2.2 to clean the web page before build the DOM-tree.	It uses the htmlcleaner-2.2 and new cleaner procedure to clean the comments, scripts, style sheets, and metadata inside the webpage.
It uses the concepts of level and non-level data blocks definitions to identify data blocks during traversing the DOM-Tree which is a very costly process that requires checking every tag node in DOM-Tree.	It proposes a new definition for both data block and data region and proposes a new efficient similarity matching technique of DOM-Tree tag nodes to locate the data blocks and data regions.
It stores the extracted data contents in relational database (RDB) which is neither historical nor object-oriented.	It stores the extracted data contents in object-oriented data warehouse (OODWH) which is more appropriate and beneficial due to the nature of the extracted data contents.

Table 4: Comparative analysis between WebOMiner-2 and WebOMiner-3.

4.2 Empirical Evaluations

We have tested our system WebOMiner-3 system so far on four websites in the domain of B2C (e.g., Bestbuy.com, Homedepto.com, Shopxscargo.com, Factorydirect.com) and compare it with the result given by (Harunorrashid, 2012) for the same four websites. WebOMiner is implemented in Java programming language. We then run our system using 64-bit Windows 7 operating system at Intel core i5 3.1 GHz, 4 GB RAM Dell machine for each of the these four websites for empirical evaluation of our WebOMiner-3. We use the standard precision and recall and error rate measures to evaluate

WebOMiner-3 compare to WebOMiner and WebOMiner-2. Recall is computed as the average percentage of total number of correct retrieved data tuples by the total number of existing data tuples in the web page as shown in formula 1 below. Precision is computed as the average percentage of total number of correct retrieved data tuples by the total number of retrieved data tuples in the web page as shown in formula 2 below. Error rate is measured as the percentage number of failed data tuples by the total number of existing data tuples in the web page as shown in formula 3 below.

$$\boxed{Recall = \frac{Total\ Correct\ Retrived\ DT}{Total\ Existing\ DT}}$$ **......................Formula 1**

$$\boxed{Precision = \frac{Total\ Correct\ Retrived\ DT}{Total\ Retrived\ DT}}$$ **..................... Formula 2**

$$\boxed{Error\ Rate = \frac{Total\ Failed\ DT}{Total\ Existing\ DT}}$$ **..................... Formula 3**

The comparative result between our system and WebOMiner and WebOMiner-2 is tabulated in table 5 below:

Website	Existed Data Records					Extracted Data Records (WebOMiner-3)			
	Product	List	Text	Noise	Total	Correct	Wrong	Missing	Failed
Bestbuy.com	12	31	1	3	47	43	0	0	4
Homedepto	24	11	5	0	40	32	0	0	8
Shopxscargo	6	27	2	4	39	34	0	0	5
Factorydirect	30	23	0	4	57	35	0	0	22
Recall						79%			
Precision						100%			
Error Rate						21%			

Table 5: Experimental results showing extraction of data records from web pages.

4.3 Experimental Results

The main purpose of experimental results is to measure the performance of WebOMiner-3 system. Table 5 shows the results of WebOminer-3 system measured by recall, precision and error rate. We have taken one page of each website for experiment and data records in columns show different types of data records. Total column represents the total number of data records in the given web page. WebOMiner-3 is able to extract different types of data records from different websites correctly with recall 79% and precision 100%. WebOMiner-3 failed to extract 39 out of 183 are existed in four web pages from different websites and achieved low recall value due to the tag nodes patterns of data blocks similarity matching technique which WebOMiner-3 uses to identify and extract data blocks from DOM-Tree. The new tag nodes similarity technique matches the tag nodes string patterns of data blocks based on the exact matching of patterns to identify the similar data blocks, so the WebOMiner-3 does not take into the consideration any degree of similarity between the data blocks and may lose some similar (not matched) data blocks. Figure 67 shows the recall of WebOminer-3 system for each website has been targeted in the experiments.

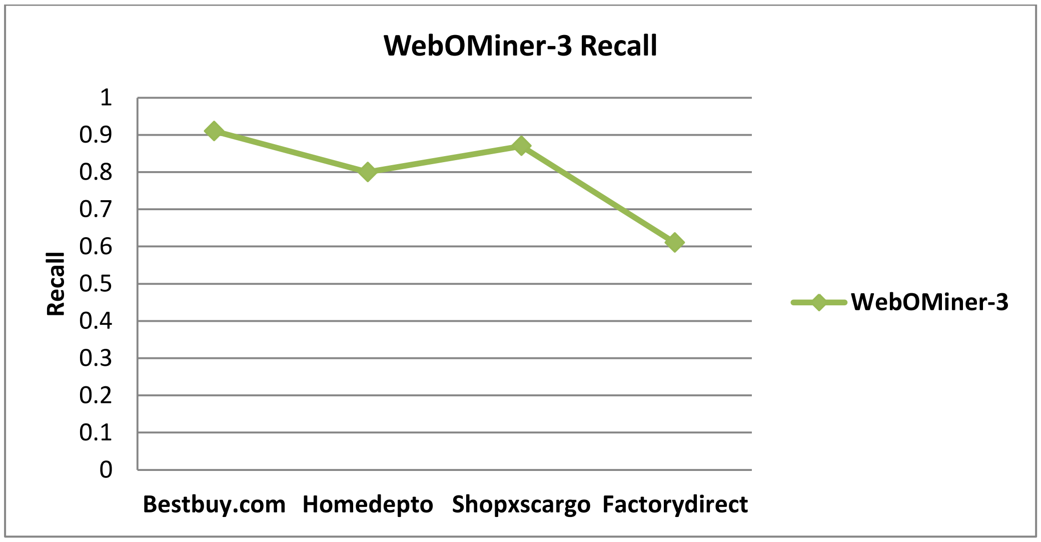


Figure 67: WebOMiner-3 Recall.

As shown in figure 67 that the WebOMiner-3 has better recall in all the websites which were included in the experiments except (Shopxscargo)'s website. For the (Bestbuy) website the recall for WebOMiner-3 is 91%. For the (Homedepto) website the recall for WebOMiner-3 is 80%. For (Shopxscargo) the recall for WebOMiner-3 is 87%. For the (Factorydirect) website the recall for WebOMiner-3 is 60%. Figure 68 shows the error rate of WebOMiner-3 in each website of the experiments. As shown in figure 68, for the (Bestbuy) website the error rate is 8%. For the (Homedepto) website the error rate for WebOMiner-3 is 20. For (Shopxscargo) the error rate for WebOMiner-3 is 13%. For the (Factorydirect) website the error rate for is 39%.

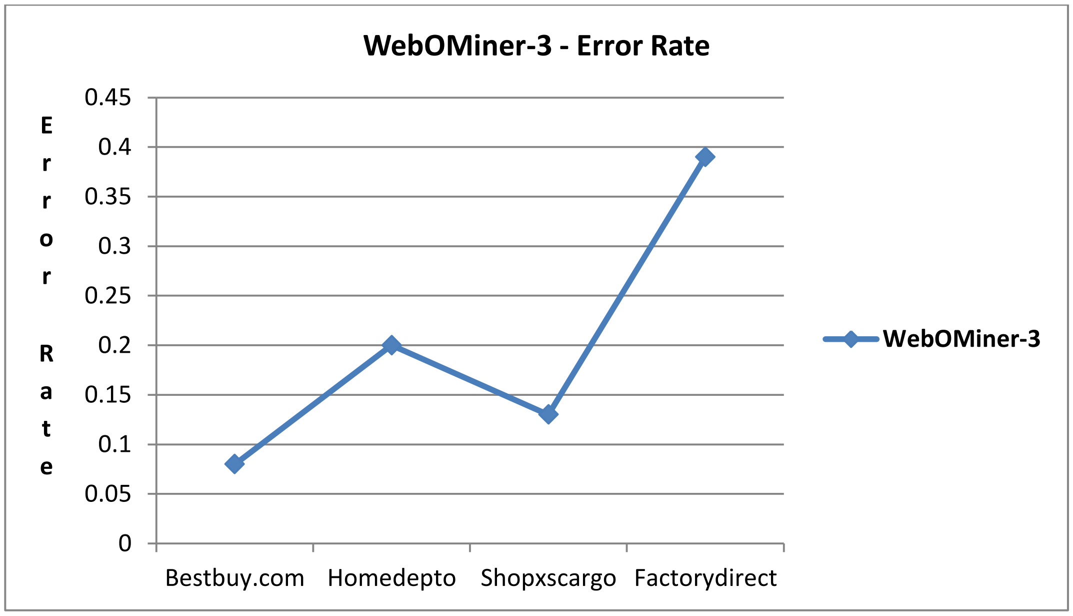


Figure 68: Error rate of WebOMiner-3.

Figure 69 shows the execution time of WebOMiner-3 for each individual website is involved in the experiments. The execution time includes the extraction process starting from the crawling step until the data records are inserted in the OODWH. The results shown that the average execution time of WebOMiner-3 is 21.8 seconds. As shown in figure 69 that the execution time of WebOMiner-3 to extract data tuples from (CompUSA)'s website is 29 seconds, 23 seconds for (Bestbuy), 17 seconds for (Homedepto), 11 seconds for (Shopxscargo), and 29 seconds for (Factorydirect)'s website. Actually, the execution time results of WebOMiner-3 is reasonable and gives a positive pointer to the good performance of the system.

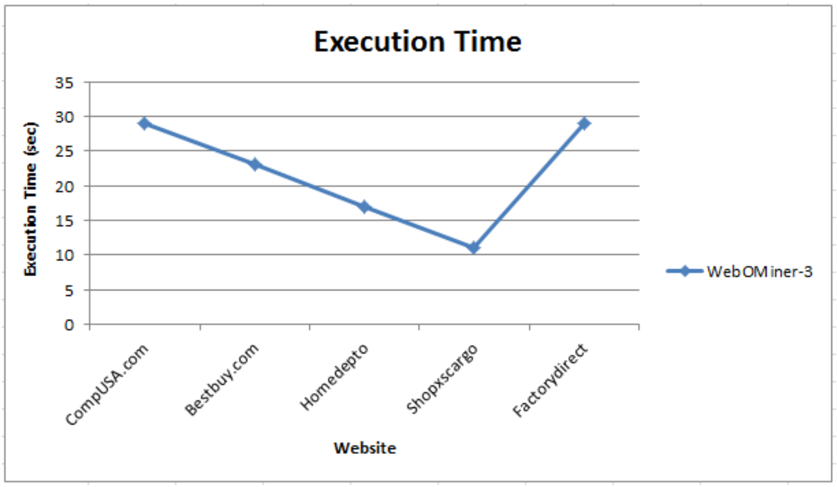


Figure 69: Execution time of WebOMiner-3 different websites.

CHAPTER 5 - Conclusion and Future Work

This thesis extends the work by Mutsuddy (2012); Ezeife and Mutsuddy (2013) and Harunorrashid (2012), to automate the extraction process of structured data records from list and detailed web pages in the domain of B2C websites, and store the extracted data tuples in historical object-oriented data warehouse for comparative shopping and eservices. The new version of the extraction system is called WebOMiner-3 and consists of five compatible modules: crawler, cleaner, extractor, frequent objects mining, and object-oriented data warehouse and integration modules. We introduced a new automatic technique based on the tag nodes similarity to extract data contents from DOM-Tree and mine these contents to build the NFA structure for each data record type to identify the extracted data tuples and store them in the correct position in object-oriented data warehouse (OODWH). The generated NFA structure helps to identify the basic structure of data warehouse which stores information about structured data tuples from different web pages in historical manner. The crawler module takes the URL of the intended web page to extract data records from it and downloads the page into the local machine; the cleaner module cleans all the comments, style and java script contents from the page. The extractor module represents the web page as DOM-Tree logical structure and then extracts the tag nodes from the tree based on our technique tag nodes similarity matching and store all the extracted data contents in array data structure. The frequent object mining module automatically builds the NFA structure for each data record type based on the frequency and sequence of data objects of data tuples. The object-oriented data warehouse module builds the structure of the data warehouse and inserts the data tuples from contents array into the right tables of data warehouse. The experiment results shown that WebOMiner-3 behaved well. The results were measured in terms of recall, precision and error rate. The results show that the recall of WebOMiner-3 is 79% and the precision is equal to 100%. The error rate for WebOMiner-3 equals to 21%.

5.1 Future Work

Although the improvements which WebOMiner-3 has done over the previous two systems WebOMiner and WebOMiner-2 in the field of object-oriented web contents mining, still there are some improvements that can be done to improve the accuracy and the performance of WebOMiner-3. Actually, the WebOMiner-3 still needs more modifications on the cleaner module to clean the contents of web page before the extraction process starts which will have positive effect on the accuracy of the system. Also the extraction module needs more optimized comparison technique based on the similarity rather than the exact matching between the string patterns of data blocks to identify them. Any similarity technique can be used to compare the tag nodes of DOM-Tree to correctly identify deeper data records inside the webpage and increase the accuracy of the extraction process which will increase the recall of WebOminer-3 system. There is a big room for the improvements of object-oriented data warehouse and integration module. The concept of object-oriented needs to be fully applied and takes more advantages of it. Finally, many of data mining techniques such as classification, frequent patterns and association rules can be applied on the OODWH to benefit the e-market and conclude future decisions.

REFERENCES

1. Agrawal R., and R. Srikant: (1995) Mining sequential patterns. In Proceedings of the 11th International Conference on Data Engineering, pp. 3–14.

2. Annoni, E. and Ezeife, C. (2009). Modeling web documents as objects for automatic web content extraction object-oriented web data model. In Proceedings of the International Conference on Enterprise Information Systems. pp. 91- 100, May 6-10, Milan, Italy.

3. Arocena, G. and Mendelzon, A. (1998). WebOQL: Restructuring documents, databases and webs. Proceedings of the 14th IEEE International Conference on Data Engineering (ICDE), pp. 24-33, Orlando, Florida.

4. Baumgartner, R., Flesca, S., and Gottlob, G. (2001). Visual web information extraction with lixto. In Proceedings of the 27th International Conference on Very Large Data Bases. VLDB'01. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 119-128.

5. Borges, J. and Levene, M. (1999). Data mining of user navigation patterns. In Proceedings of the WBBKDD'99 Workshop on Web Usage Analysis and User Profiling, August 15, 1999, San Diego, CA, USA, pp. 31-36.

6. Bornhvd, C. and Buchmann, A. (1999). A prototype for metadata-based integration of internet sources. In Advanced Information Systems Engineering, M. Jarke and A. Oberweis, Eds. Lecture Notes in Computer Science, vol. 1626. Springer Berlin / Heidelberg, 439-445.

7. Buchner, A. G. and Mulvenna, M. D. (1998). Discovering internet marketing intelligence through online analytical web usage mining. SIGMOD Rec. Vol.27, No.4, 54-61.

8. Chang, C.H., and Lui, S.C. (2001). IEPAD: information extraction based on pattern discovery. Proceedings of the 10th Int'l Conf. World Wide Web (WWW), pp. 223-231.

9. Chowdhury, G. (2010). Introduction to Modern Information Retrieval, Third Edition, 3rd ed. Facet Publishing.

10. Crescenzi, V., Mecca, G., and Merialdo, P. (2002). ROADRUNNER: automatic data extraction from data-intensive web sites. In Proceedings of the 2002 ACM SIGMOD international conference on Management of data. NY, USA, pp. 624-624.

11. Demner-Fushman, D. (2006). Complex question answering based on a semantic domain model of clinical medicine. Ph.D. thesis, University of Maryland at College Park.

12. Ezeife, C.I., and Mutsuddy, T. (2013). Towards comparative mining of web document objects with NFA: WebOMiner system, to appear in the International Journal of Data Warehousing and Mining (IJDWM), a 2013 volume.

13. Etzioni, O. (1996). The world-wide web: quagmire or gold mine? Commun. ACM 39, 11, 65-68.

14. Ezeife, C. I. and Zhang, D. (2009). Tidfp: Mining frequent patterns in different databases with transaction id. In Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery. DaWaK '09. Springer-Verlag, Berlin, Heidelberg, pp. 125-137.

15. Gupta, S., Kaiser, G., and Stolfo, S. (2005). Extracting context to improve accuracy for html content extraction. In Special interest tracks and posters of the 14th international conference on World Wide Web. WWW '05. ACM, NY, USA, pp. 1114-1115.

16. Hammer, J., Garcia-Molina, H., Cho, J., Aranha, R., and Crespo, A. (1997). Extracting semi structured information from the web. Technical Report 1997-38, Stanford InfoLab.

17. Han, J. and Kamber, M. (2000). Data mining: concepts and techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

18. Harunorrashid, M. (2012). Mining multiple web sources using finite state automata. Master's thesis, School of Computer Science, University of Windsor.

19. H.Inmon, W. (2005). Building the data warehouse. Addison-Wesley, Reading, Mass.

20. Hsu,C.N. and Dung,M.T. (1998). Generating finite-state transducers for semi-structured data extraction from the Web. Information Systems, Volume 23, Issue 8, pp. 521-538.

21. Kao, A. and Poteet, S. (2005). Text mining and natural language processing: introduction for the special issue. SIGKDD Explor. Newsl. 7, 1 (June), 1-2.

22. Kemper, A., and Moerkotte G.(1994). Object-oriented database management. Published by Prentice-Hall Inc, ISBN: 0-13-629239-9.

23. Kim, W. (1990). Object-oriented databases: definition and research directions. Knowledge and Data Engineering, IEEE Transactions on 2, 3 (sep), 327-341.

24. Kosala, R. and Blockeel, H. (2000). Web mining research: a survey. SIGKDD Explor. Newsl. 2, 1-15.

25. Kushmerick, N., Weld, D., and Doorenbos, R. (1997). Wrapper induction for information extraction. Proceedings of the Fifteenth International Conference on Artificial Intelligence (IJCAI), pp. 729-735.

26. Laender, A. H. F., Ribeiro-Neto, B., and da Silva, A. S. (2002). Debye - date extraction by example. Data Knowl. Eng. 40, 2 (Feb.), 121-154.

27. Laender, A. H. F., Ribeiro-Neto, B. A., da Silva, A. S., and Teixeira, J. S. (2002). A brief survey of web data extraction tools. SIGMOD Rec. 31, 84-93.

28. Levering, R. and Cutler, M. (2006). The portrait of a common html web page. In Proceedings of the 2006 ACM symposium on Document engineering. DocEng '06. ACM, NY, USA, pp. 198-204.

29. Li, J. and Ezeife, C. (2006). Cleaning web pages for effective web content mining. In proceedings of the 17th International Conference on Databases and Expert Systems Applications, DEXA 2006, Krakow, Polland, Sept 4-8, published in LNCS, pp. 560-571, Springer Verlag.

30. Li, Z. and Ng, W. K. (2004). Wiccap: from semi-structured data to structured data. In proceeding of 11th IEEE international conference and workshop on Engineering and Computer based Systems. Brno, Czech Republic, May 24-27: pp. 86-93.

31. Liu, B. (2006). Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data (Data-Centric Systems and Applications). Springer-Verlag New York, Inc., Secaucus, NJ, USA.

32. Liu, B., Grossman, R., and Zhai, Y. (2003). Mining data records in web pages. In Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '03. ACM, NY, USA, pp. 601-606.

33. Liu, B. and Zhai, Y. (2005). Net a system for extracting web data from at and nested data records. In Web Information Systems Engineering WISE 2005, vol. 3806. Springer Berlin / Heidelberg, 487-495.

34. Liu, L., Pu, C., and Han, W. (2000). XWRAP: an XML-enabled wrapper construction system for web information sources. Data Engineering, 2000. Proceedings 16th international conference on , Vol., No., pp.611-621.

35. Madria, S., Bhowmick, S., Ng, W., and Lim, E. (1999). Research issues in web data mining. In Data Warehousing and Knowledge Discovery. Lecture Notes in Computer Science, vol. 1676. Springer Berlin / Heidelberg, pp. 805-805.

36. Morrison, D. R. (1968). Patricia practical algorithm to retrieve information coded in alphanumeric. J. ACM 15, 4 (Oct.), pp. 514-534.

37. Muslea, I., Minton, S., and Knoblock, C. (1999). A hierarchical approach to wrapper induction. In Proceedings of the third annual conference on Autonomous Agents. AGENTS '99. ACM, NY, USA, pp. 190-197.

38. Mutsuddy, T. (2010). Towards comparative web content mining using object oriented Model. Master’s thesis, School of Computer Science, University of Windsor.

39. Oh, J., Lee, J., Kote, S., and Bandi, B. (2003). Multimedia data mining framework for raw video sequences. Lecture Notes in Computer Science, vol. 2797. Springer Berlin / Heidelberg, pp. 18-35.

40. Raposo, J., Pan, A., Alvarez, M., Hidalgo, J., and Vina, A. (2002). The wargo system: semi-automatic wrapper generation in presence of complex data access modes. In proceedings of 13th International Workshop on Database and Expert Systems Applications. pp. 313-317.

41. Ristad, E. S. and Yianilos, P. N. (1996). Learning string edit distance. Tech. Rep. CS-TR-532-96, Princeton University, Department of Computer Science.

42. Sahuguet, A. and Azavant, F. (1999). Wysiwyg web wrapper factory (w4f). In proceedings of Toronto, ON, Canada.

43. Sebastiani, F. (2002). Machine learning in automated text categorization. ACM Comput. Surv. 34, 1 (Mar.), 1-47.

44. Soderland, S.(1999). Learning information extraction rules for semi-structured and free text. Journal of Machine Learning, 34(1-3): pp. 233-272.

45. Srivastava, J., Cooley, R., Deshpande, M., and Tan, P.-N. (2000). Web usage mining: discovery and applications of usage patterns from web data. SIGKDD Explor. Newsl. 1, 2 (Jan.), pp. 12-23.

46. Zhai, Y. and Liu, B. (2005). Web data extraction based on partial tree alignment. In Proceedings of the 14th international conference on World Wide Web. WWW '05. ACM, NY,USA, pp. 76-85.

47. Zhai, Y. and Liu, B. (2007). Extracting Web Data Using Instance-Based Learning. Lecture notes in computer science, Springer, vol. 10(2) pp. 113-132.

48. Zhang, D. (2011). Mining Multiple Related Tables using Object-Oriented Model. Master’s thesis, School of Computer Science, University of Windsor.

VITA AUCTORIS

NAME:	Yanal Alahmad
PLACE OF BIRTH:	Ramtha, Jordan
YEAR OF BIRTH:	1982
EDUCATION:	Al al-Bayt University, B.Sc., Mafreq, Jordan, 2004
	Jordan University of Science and Technology, M.Sc., Irbid, Jordan, 2007.
	University of Windsor, M.Sc., Windsor, ON, 2012