An Automated Approach for Retrieving Hierarchical Data from HTML Tables

Seung-Jin Lim
Yiu-Kai Ng
Computer Science Department
Brigham Young University
Provo, Utah 84602, U.S.A.
Email: {ng,sjlim}@cs.byu.edu

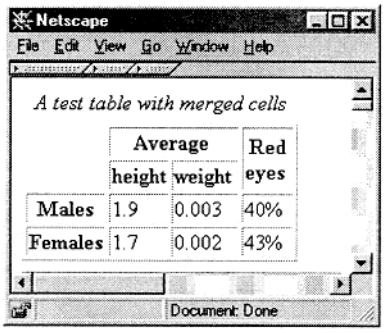
Abstract

Among the HTML elements, HTML tables [RHJ98] encapsulate hierarchically structured data (hierarchical data in short) in a tabular structure. HTML tables do not come with a rigid schema and almost any forms of two-dimensional tables are acceptable according to the HTML grammar. This relaxation complicates the process of retrieving hierarchical data from HTML tables. In this paper, we propose an automated approach for retrieving hierarchical data from HTML tables. The proposed approach constructs the *content tree* of an HTML table, which captures the *intended hierarchy* of the data content of the table, without requiring the internal structure of the table to be known beforehand. Also, the user of the content tree does not deal with HTML tags while retrieving the desired data from the content tree. Our approach can be employed by (i) a query language written for retrieving hierarchically structured data, extracted from either the contents of HTML tables or other sources, (ii) a processor for converting HTML tables to XML documents, and (iii) a data warehousing repository for collecting hierarchical data from HTML tables and storing materialized views of the tables. The time complexity of the proposed retrieval approach is proportional to the number of HTML elements in an HTML table.

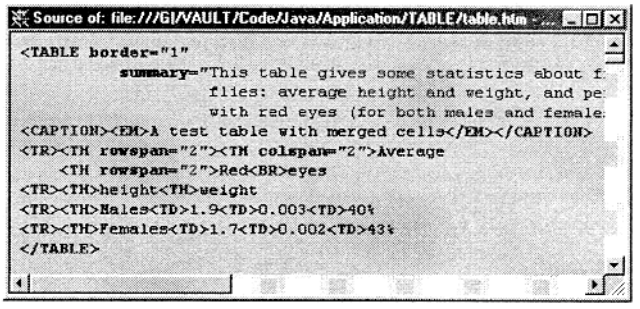
1 Introduction

We are interested in automated data retrieval tools and are particularly interested in retrieving textual data that are hierarchically structured in HTML documents [RHJ98]. Among the information that we deal with on a daily basis, we realize that there are vast amounts of data that are hierarchically related. Examples of data in this category are "the average height of males of merged cells," "homes listed between $150,000 and $180,000, having at least 3 bedrooms, and located in North Orem, Utah" and so forth. This type of information usually can be inferred from hierarchically structured data.

Although the primary purpose of HTML is to define appearance of data in a browser, we notice that the creator of an HTML document often imposes, either intentionally or unintentionally, certain hierarchies among the data contents in the document by using block markups. We consider two types of data hierarchies, the syntactic hierarchy and the intended hierarchy, in HTML documents. The *syntactic hierarchy* of an HTML document H, which can be recognized by using the HTML grammar, captures the hierarchy of *tags* and *data contents* in H as defined by HTML. For example, consider the HTML table T appeared in [RHJ98] and its source code as shown in Figure 1(b). The syntactic hierarchy of T is shown in Figure 2, which will be further discussed in detail in Section 2. The *intended hierarchy*, as we call it, is the hierarchy of *data contents* in an HTML document with the exclusion of all the tags in the document. Consider Figure 1(a) for an illustration of the intended hierarchy. It is clear that the number *'1.9'* in the table is the `average height` of `males`. Furthermore, we can infer the following information: the `average height` of `males` of `merged cells` is *1.9*; the `average height` of `females` of `merged cells` is



(a) Rendered table in Netscape



(b) The source code of the table in Figure 1(a)

Figure 1: A sample HTML table

CIKM '99 11/99 Kansas City, MO, USA

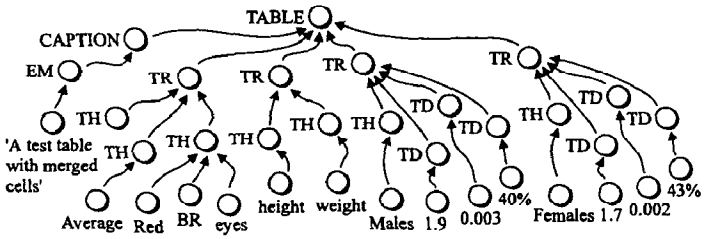


Figure 2: The syntactic hierarchy of the HTML table in Figure 1

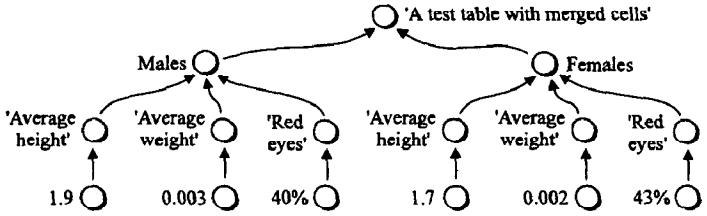


Figure 3: The intended hierarchy of the data contents of the HTML table in Figure 1

1.7; the `average weight` of `males` of `merged cells` is *0.003*; and so forth. Figure 3 shows the intended hierarchy of the table cells by transforming '*A of B*' in any of the inferred information listed above to '*B is the parent node of A*' in the hierarchy.

We now compare the two types of hierarchies using the HTML table in Figure 1. It is easy to see that the hierarchical relationships among the data components `average height`, `males`, and `merged cells` are not precisely preserved in the syntactic hierarchy (as shown in Figure 2). It is difficult to determine the relationships among these data components without considering the meaning of their ancestors (such as `TR`, `TH`, `TD`, and `TABLE`) in the hierarchy. Existing approaches [AMM97, HGMC+97, AM98] for extracting hierarchical data are based on syntactic hierarchies. In these approaches, the source documents must be analyzed before hierarchical data are extracted [AMM97] and/or HTML tags in the documents are extensibly used [HGMC+97, AM98] to determine their data hierarchies. In contrast, the intended hierarchy of an HTML document H (e.g., the document in which the table in Figure 1 is embedded) preserves the hierarchical relationships of the data contents in H such that the hierarchy of the data contents is determined by the associated edges without using tags in H.

In this paper, we present an approach to determine the intended hierarchy of the data contents in an HTML table[1] that contains heading (TH) and cell data elements (TD). Table-related HTML elements are often chosen by document creators for encapsulating hierarchical data and are increasingly being used. Our recent survey shows that 52% of the sampled HTML documents include some tables. In the survey, we chose a number of Web sites of common interest and fetched more than 30,000 documents which were linked from the initial HTML documents within 4 hops, where the number of hops is the number of hyperlinks between an initial document and a linked document.

Given an HTML table T as input, our retrieval approach generates a tree, called the *content tree* (CT), of T as output. CT captures the intended hierarchy of the data contents in T and is generated without requiring the internal structure of T to be known beforehand at the source code level (such as which HTML markups are used and how they are used) or at the conceptual level (such as how many columns and rows are declared in T, whether headings and the caption of T exist, etc.). Determining the data hierarchy of an HTML table has not been addressed in the literature, and CT is distinct from other approaches [KS95, ACC+97, MM97, AMM97, AM98] that are based on the syntactic hierarchy for extracting hierarchical data from HTML documents. Existing data collection methods, such as wrappers and integrators for data warehousing systems, which frequently access a large number of HTML tables for extracting structured data, such as stock price quotes, weather information, medical data, law suites, etc., can benefit from our automated approach of data acquisition. Furthermore, the content tree of an HTML table can be used for converting the HTML table to an XML document. We demonstrate how to obtain an XML document using a content tree in Section 4.2.

We proceed to present our results as follows. In Section 2, we introduce a data model for representing hierarchical data in HTML documents. In Section 3, we present our approach for constructing the content tree of an HTML table and provide the complexity analysis of the process. In Section 4, we discuss a few applications of content trees which include using hierarchy-based queries for retrieving hierarchical data of interest from an HTML table, conversion of HTML tables to XML documents, and the design of a data warehouse using content trees. In Section 5, we give the conclusions.

2 The data model: semistructured data tree

Since HTML documents do not come with a rigid structure, they are *semistructured* in nature. Any data can be conceived as semistructured data if they do not have a rigid structure or schema, nor unstructured. If a Web document D conforms to the HTML specification, we call D a *valid* HTML document. A valid HTML document consists of a number of HTML elements. A typical *HTML element* is delimited by its *start-tag* and *end-tag*, and contains *content* that appears between its start-tag and end-tag. For some HTML elements, their end-tags are implicit and/or their contents do not exist. An HTML element may contain other tags (i.e., an HTML element can be *nested*) or data characters. We call the latter *data content*.

Our semistructured data model is called semistructured data tree (SDT), which is a tree-like, conceptual representation (view) of the underlying semistructured data. The core constructs of an SDT are (i) *objects*, which are defined over the set of all possible `strings`, denoted by Σ^*, and (ii) *dependency constraints* among the objects.

Definition 1 An *object* o in an HTML document D is a 3-tuple $\langle name, attribute\ list, identifier \rangle$, which is either a start-tag (called a *tag object*) or data content (called a *data object*) in D, where

- $name$[2] $\in \Sigma^*$ is a meaningful textual representation of o in D. If o is a tag object, the name of o is the name of the corresponding HTML tag. If o is a data object, the name of o is the corresponding data characters.
- *attribute list* $\subset \Sigma^*$ is a finite (possibly empty) list of *attributes*, each of which is of the form *attribute_name* = *value*.
- $identifier \in \Sigma^*$ is a non-empty `string` which uniquely identifies o in D. □

[1] An approach for determining the hierarchy of the data contained in generic HTML elements, other than tables, can be found in [LN98].

[2] A name may include spaces and dots. Any name including spaces and/or dots is enclosed by single quotes.

Definition 2 Given an HTML document D, an object o_1 *directly depends on* another object o_2, denoted $o_1 \leftarrow o_2$, if o_1 immediately *contains* o_2, i.e., o_2 is an immediate *content* of o_1. □

Example 1 Consider the content of the HTML table in Figure 1(b). <TABLE>, <CAPTION>, and <TR> with names *TABLE*, *CAPTION*, and *TR*, respectively, are tag objects. `'A test table with merged cells'` is the data content of the tags <EM> and </EM>, and is a data object. Since <TABLE> contains <CAPTION>, $TABLE \leftarrow CAPTION$. Furthermore, $TABLE \leftarrow CAPTION \leftarrow EM \leftarrow$ *'A test table with merged cells'*. □

Semistructured data, in its simplest notion, is a set of objects with associated dependency constraints, and so are HTML documents. Note that the syntactic and intended hierarchies shown in Figures 2 and 3, respectively, are the tree representations of their respective objects and associated dependency constraints. The notion of long names defined below represents a chain of objects with their dependency constraints lexically.

Definition 3 Given $o_1 \leftarrow o_2 \leftarrow \ldots \leftarrow o_n$, the *long name* of o_n, denoted $\mathcal{L}_{o_n}$, is the concatenation of the names of objects $o_1, o_2, \ldots, o_n$ in the form $o_1.o_2.\ldots.o_n$. (The dot '.' notation between two object names is called the *dot* operator which is used as a separator of two names.) The *path expression* of objects o_i and o_j $(1 \leq i \leq j \leq n)$, denoted by $pe(o_i, o_j)$, is a substring of $\mathcal{L}_{o_n}$ of the form $o_i.\ldots.o_j$. □

By using the notion of long names, semistructured data D can be expressed as an ordered list of long names of objects in D. In conjunction with long names, we adopt the notation $A.(B, C)$ for an ordered list $(A.B, A.C)$.

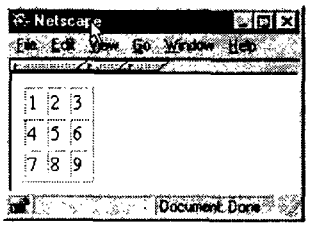
3 An approach for constructing the intended hierarchy

In this section, we present our approach for constructing the intended hierarchy of the data contents in a given HTML table. Prior to presenting our approach, we first discuss table-specific HTML elements.

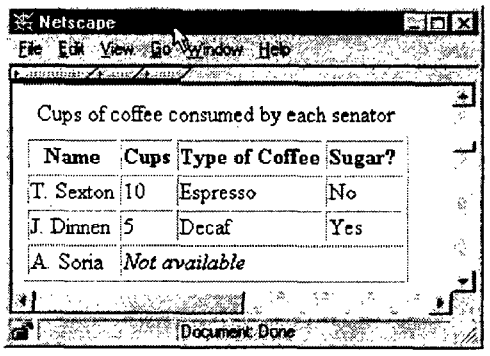
3.1 Table-specific elements

Among the table-specific elements, TR determines the number of rows, whereas TH and TD determine the number of columns in an HTML table (table in short). A TH element is used for declaring one or more headings, whereas a TD element is used for asserting data of a table cell. The data contents of TD elements are called *table data* since they are the data instances in their respective cells in an HTML table. In contrast, the data contents of TH elements are headings, i.e., column headings or row headings that are not considered as table data. However, the data content of either element renders the content of its respective cell by a visual HTML user agent, such as a Web browser. (A visual HTML user agent, however, may not explicitly indicate that one cell is a heading cell while another is a data cell.) Besides TH and TD, THEAD (resp. TBODY) can be used for implicitly designating certain rows as headings (resp. data cells).

The experimental survey, as mentioned in Section 1, also revealed the following regarding the use of table-specific elements in HTML documents: (i) A typical HTML table has at least one heading row at the top of the table and at least one heading column on the left, such as the table shown in Figure 1(a). We call this type of tables *column-row-wise*. (ii)



(a) An HTML table without heading



(b) An HTML table with a first-row heading

Figure 4: Two other prominent types of HTML tables

Another typical table contains at least one heading row at the top of the table, such as the table shown in Figure 4(b) in [RHJ98]. We call this type of tables *column-wise*. In a column-wise table, the heading rows yield the schema of the table. (iii) Other than these two types of tables, we notice that a large number of tables do not make use of table-specific elements other than TABLE and TD, as the one rendered in Figure 4(a). In such a table, every cell is in fact a data cell (according to the HTML grammar), and the intended hierarchy of the data cells is very difficult, if not impossible, to be determined by an automaton at the source code level using the HTML grammar alone. We draw from our survey and analysis a conclusion that the creator of this type of tables often implicitly designates the first row as a heading. Hence, we treat this type of tables as column-wise.

Among the table-specific elements, two attributes of TH and TD, namely ROWSPAN and COLSPAN, play significant roles in determinating the data hierarchy of an HTML table. To better understand the roles of these attributes, consider Figure 2. The table, as graphically captured in the figure, has four rows (i.e., four TRs) as rendered in Figure 1(a). Note that the first TR contains three TH elements, implying that the row is a heading which includes three cells (i.e., three columns). The second row, on the other hand, contains two TH elements, implying that there are two heading cells, whereas each of the last two TRs contains one TH and three TDs, implying that there are one heading cell on the left and three data cells in each row underneath the heading rows. Apparently, the numbers of columns of the first two rows are not equal, nor with that of the last two rows either. It is not clear which cell in one row belongs to the same column with a cell in another row unless we consider the attributes of the corresponding objects (for the definition on attributes of an object, see Definition 1). ROWSPAN and COLSPAN are the attributes that remedy the mismatch of the number of columns among different rows in a table. When a TH or TD includes

ROWSPAN="n" (COLSPAN="n," respectively), the associated cell is supposed to span n rows downward (n columns to the right, respectively). These attributes are taken into consideration in the construction process of the content tree of a given HTML table. In this paper we assume that data are presented in an HTML table from left to right.

3.2 Content trees

In order to create the content tree (CT), which is an enhancement of the syntactic tree, of any HTML table T, we first determine the hierarchical dependencies among the data contents in T (instead of the hierarchy of the data contents and tags in T). Once the hierarchical dependencies among the data contents in T are determined, we retain only data objects in CT and exclude all the tag objects. This is because tags are markups for defining the rendered appearance of data in a visual HTML user agent and are invisible to the user, and hence they are excluded from CT.

A *content tree* is defined on top of the notion of *pseudo-table* since the properties of a pseudo-table are easy to understand. A pseudo-table can be considered as a special type of HTML tables that can be used for expressing either a column-row-wise table or a column-wise table. Our strategy in constructing the content tree of an HTML table T is to first map T to a pseudo-table and then obtain the content tree of T from the pseudo-table. Prior to defining content tree, we first introduce pseudo-tables.

Definition 4 A *pseudo-table* $T = \{(a_{1,1}, \ldots, a_{1,n}), (a_{2,1}, \ldots, a_{2,n}), \ldots, (a_{m,1}, \ldots, a_{m,n})\}$ with column headings $C_1, \ldots, C_n$ and caption C, is a two-dimensional table, where each column heading C_i $(1 \leq i \leq n)$ or table data $a_{i,j}$ $(1 \leq i \leq m, 1 \leq j \leq n)$ may be null. If a column heading or table data o is null, then the name of the object representing o is the empty string. Also, data values of row i and row j of the first column in T are different if $i \neq j$. □

Note that dependency constraints of HTML objects in a syntactic hierarchy are determined by the container-content relationships of tags and data contents according to the HTML grammar, as illustrated in Example 1 and Figure 2. We define the notion of dependency for pseudo-tables from a different perspective since there are no tags in a pseudo-table. Dependency constraints on pseudo-tables are defined over the caption, column headings and table data, and dependency constraints among these objects are given in the following definition. Since column headings and table data may be null in a pseudo-table, we consider a special case of dependency constraints: given a dependency constraint $o_1 \leftarrow o_2 \leftarrow o_3$, where o_i $(1 \leq i \leq 3)$ is either a column heading or table data, the dependency constraint is reduced to $o_1 \leftarrow o_3$ if the name of o_2 is the empty string.

Definition 5 Given an n-ary pseudo-table $T = \{(a_{1,1}, \ldots, a_{1,n}), (a_{2,1}, \ldots, a_{2,n}), \ldots, (a_{m,1}, \ldots, a_{m,n})\}$ with column headings $C_1, \ldots, C_n$ and caption C, the content tree $CT = (V, E, g)$ of T, denoted by CT_T, is a directed tree, where

- $V_R \in V$ is the root node of CT_T, labeled by C, and each node $v \in V$, other than V_R, denotes a non-empty $a_{i,j}$ (C_j, respectively) $(1 \leq i \leq m, 1 \leq j \leq n)$ of T, and is labeled by $a_{i,j}$ (C_j, respectively).
- E is a finite set of *directed* edges.
- g: $E \rightarrow V \times V$ is a function such that $v_2 \leftarrow v_1$ if $g(e) = (v_1, v_2)$, and $V_R \leftarrow C_1 \leftarrow a_{i,1} \leftarrow C_j \leftarrow a_{i,j}$ $(1 \leq i \leq m, 2 \leq j \leq n)$ hold. □

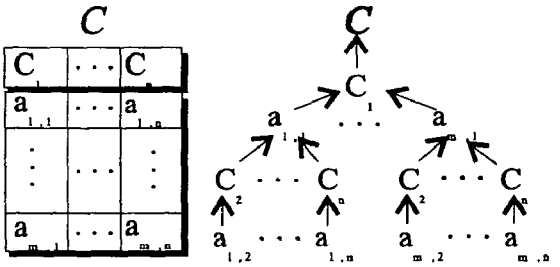


Figure 5: A pseudo-table T and its corresponding content tree CT_T

Since the caption of a pseudo-table T provides a short description on what the table is about, we choose the caption as the root node of the corresponding CT_T to assert the content of the table. In the definition of content tree, CT_T contains subtrees rooted at $a_{1,1}, \ldots, a_{m,1}$ with the constraints V_R $(C) \leftarrow C_1 \leftarrow a_{i,1}$ $(1 \leq i \leq m)$. This is because each row can be uniquely identified from the other rows by the first column (i.e., $a_{i,1}$) in T (see Definition 4). Figure 5 shows a pseudo-table and its corresponding content tree constructed according to Definition 5.

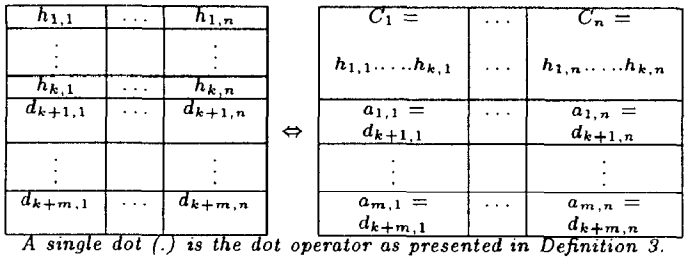
Definition 5 captures the process of constructing the content tree from a pseudo-table. As mentioned earlier, in order to construct the content tree of an HTML table T, we first obtain the corresponding pseudo-table from T and then create the content tree from the pseudo-tree. In the following sections, we discuss how to obtain each row, i.e., $(a_{i,1}, \ldots, a_{i,n})$ $(1 \leq i \leq m)$, and column, i.e., $(a_{1,j}, \ldots, a_{m,j})$ $(1 \leq j \leq n)$ from the corresponding HTML table, which is either column-wise or column-row-wise. Note that by the HTML specification, an HTML table contains at most one CAPTION element. If it exists, it becomes the caption C of the pseudo-table; otherwise, the string value "TABLE" is the caption C.

3.2.1 Column-wise tables

Consider a given column-wise HTML table $T = \{(h_{1,1}, \ldots, h_{1,n}), \ldots, (h_{k,1}, \ldots, h_{k,n}), (d_{k+1,1}, \ldots, d_{k+1,n}), \ldots, (d_{k+m,1}, \ldots, d_{k+m,n})\}$, where each $h_{i,j}$ $(1 \leq i \leq k, 1 \leq j \leq n)$ is the data content of a TH element and each $d_{i,j}$ $(k+1 \leq i \leq k+m, 1 \leq j \leq n)$ is the data content of a TD element. T, like a pseudo-table, is a collection of rows such that each data row contains the data instances corresponding to the respective column headings, and hence we treat each data row (resp. column) in T as a row (resp. a column) in a pseudo-table by mapping $d_{l,j}$ to $a_{i,j}$ in a pseudo-table such that $a_{i,j} = d_{l,j}$, $1 \leq i \leq m$, $1 \leq j \leq n$, $k+1 \leq l \leq k+m$. In addition, by the definition of a column-wise table, the first k rows specify the column headings $C_1, \ldots, C_n$ of a pseudo-table. Since the first k rows in a column-wise HTML table are headings, the first k rows of a particular column together suggest the contents of the column. Hence, we concatenate the first k rows of column j $(1 \leq j \leq n)$ and assign the concatenation to be the column heading C_j such that $C_j = concat(\ldots(concat(h_{1,j}, h_{2,j}), \ldots), h_{k,j})$, $1 \leq j \leq n$. Figure 6 illustrates how the data contents of THs and TDs in a column-wise HTML table are mapped to C_js and $a_{i,j}$s in a pseudo-table, respectively.

3.2.2 Column-row-wise tables

Consider a given column-row-wise HTML table $T = \{(hh_{1,1}, \ldots, hh_{1,n}), \ldots, (hh_{k,1}, \ldots, hh_{k,n}), (hv_{k+1,1}, \ldots, hv_{k+1,l}, d_{k+1,l+1}, \ldots, d_{k+1,l+n-1}), \ldots, (hv_{k+m,1}, \ldots, hv_{k+m,l},$



A single dot (.) is the dot operator as presented in Definition 3.

Figure 6: Mapping from a column-wise HTML table to a pseudo-table

$d_{k+m,l+1}$, ..., $d_{k+m,l+n-1}$)}, where each $hh_{i,j}$ $(1 \leq i \leq k, 1 \leq j \leq n)$ and $hv_{i,j}$ $(k+1 \leq i \leq k+m, 1 \leq j \leq l)$ is the data content of a TH element, and each $d_{i,j}$ $(k+1 \leq i \leq k+m, l+1 \leq j \leq l+n-1)$ is the data content of a TD element.

Just like column-wise tables, we concatenate the first k rows in a column-row-wise table column-by-column since they are the headings of columns in the table, and assign the jth concatenated heading to be the jth column heading C_j in the corresponding pseudo-table such that $C_j = concat(\ldots(concat(hh_{1,j}, hh_{2,j}), \ldots), hh_{k,j})$, $1 \leq j \leq n$. In addition, the first l columns of each row, starting from the $(k+1)$th row, yield the headings of the rows in this type of tables. Hence, we concatenate the first l columns of each row, starting from the $(k+1)$th row, and assign the ith concatenated heading to be the first column of the ith data row, i.e., $a_{i,1}$, such that $a_{i,1} = concat(\ldots(concat(hv_{i,1}, hv_{i,2}), \ldots), hv_{i,l})$, $k+1 \leq i \leq k+m$, and $a_{i,j} = d_{i,l+j-1}$, $k+1 \leq i \leq k+m$, $2 \leq j \leq n$. Figure 7 illustrates how the data contents of THs and TDs in a column-row-wise HTML table are mapped to C_js and $a_{i,j}$s in a pseudo-table, respectively.

3.2.3 Colspan and rowspan

As mentioned in Section 3.1, an HTML table may not have the same number of columns in each row, and COLSPANs and ROWSPANs play an important role in mapping such an HTML table to a pseudo-table. We now discuss how to manipulate COLSPANs and ROWSPANs in THs or TDs.

If a TH or TD element in an HTML table contains COLSPAN = "n," the particular cell of the TH or TD is supposed to be expanded to n columns and occupy n cells, including the current cell in the current row. Hence, at the current row, we insert $n-1$ cells to the right of the current cell and replicate the data content of the current cell $n-1$ times to the new cells since the data content of the current cell is meant to be the same over the next $n-1$ cells in the same row. ROWSPAN, however, is processed differently than COLSPAN. If a TH element contains ROWSPAN = "n," the particular cell of the TH element is supposed to be expanded to the next $n-1$ rows, and occupy n cells, including the current cell and the new $n-1$ cells in the current column. For that we insert $n-1$ cells beneath the current TH cell in the current column, and push the data content h of the current cell all the way down to the $(n-1)$th new cell, rather than replicating h to the underneath rows $n-1$ times. As a result, h appears at the $(n-1)$th new cell, and each of the cells above the $(n$-1)th new cell in the same column is left as the empty string. This is necessary for retaining the correct association among the table data across all the rows in a column while avoiding repetition of the same heading label. Recall that the concatenated headings of a column in an HTML table is mapped to a column heading in a pseudo-table. If the data content, say d, of the cell is replicated in the next $n-1$ cells vertically as we do horizontally for COLSPAN="n," the identical heading label will repeatedly appear n times in its corresponding column heading C_i of a pseudo-table, which yields $d.d.....d$, since the data content of these n cells are to be concatenated, and subsequently, in the long name of the node representing C_i in the corresponding CT. However, if ROWSPAN is contained in a TD, we insert $n-1$ new cells underneath the current TD cell, and replicate the data content of the TD to the inserted cells since each table data in different rows of the same column is meant to represent a data entry with the same content.

After COLSPANs and ROWSPANs are properly processed, the given table conforms to the definition of pseudo-table as given in Definition 5.

Cups of coffee consumed by each senator

Name	*Cups*	*Type of Coffee*	*Sugar?*
T. Sexton	10	Espresso	No
J. Dinnen	5	Decaf	Yes
A. Soria	Not available	Not available	Not available

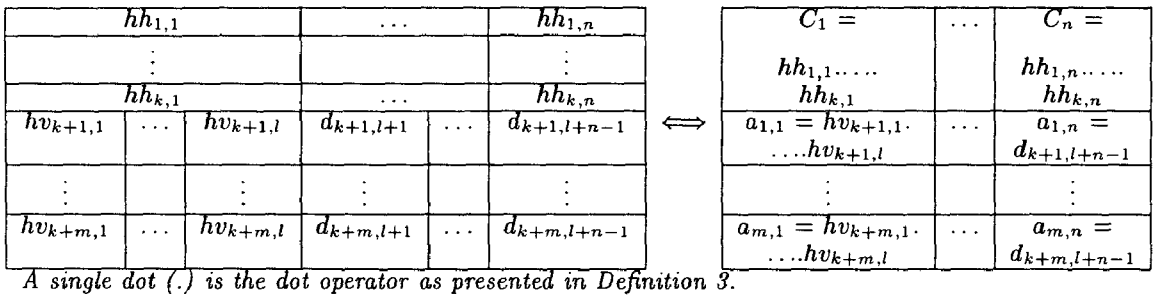
Figure 8: The pseudo-table of the HTML table in Figure 4(b)

Example 2 Consider the column-wise table in Figure 4(b) and suppose the cell at the ith row and the jth column is denoted by $cell(i,j)$. Note that "Not available" appears across the last three column cells of the forth row of the table. This is because in the source code (which is not included in this paper due to page limit) COLSPAN="3" is associated with cell(4, 2), and the data content "Not available" of cell(4, 2) is replicated to the next two inserted cells, i.e., cell(4, 3) and cell(4, 4), in its corresponding pseudo-table. The resulting table is as shown in Figure 8. □

Example 3 Consider the source code in Figure 1(b), where the first TH contains null data, the second TH contains the data content "Average," and the third TH contains the data content "Red eyes." We demonstrate the process of constructing the content tree of the HTML table as shown in Figure 1(a). Since the first and third TH both contain ROWSPAN="2", the null data of the first one is pushed down to the next row in the corresponding pseudo-table T. At a glance, it may look like that this action has no effect to the table since the pushed-down value is a null data. Indeed, with this action, the pushed-down null data is inserted into cell(2, 1) in T and subsequently the TH with *height* (*weight*, respectively) is moved to the new location which is cell(2, 2) (cell(2, 3), respectively) in T. This is desirable since the correct association among "height," "weight," and other table data are now in place in T. In addition, the second TH contains COLSPAN = "2" and subsequently its data content "Average" is replicated once to the right in the same row in T, and "Red eyes," which is originally in the third TH, is moved to the forth column and then pushed onto the forth column of the next row in T because of the attribute ROWSPAN = "2."

Note that there are two heading rows in T and each C_i $(1 \leq i \leq 4)$ is determined by the concatenation of the data contents of the two rows in the ith column (see Definition 5 for the definition of C_i). As a result of concatenating the first and second rows in T, C_1 is the empty string, C_2 = `Average.height`, C_3 = `Average.weight`, and C_4 = `'Red eyes.'` Furthermore, the caption of T is the caption of the HTML table in Figure 1. The resulting pseudo-table T is as shown in Figure 9.

We now map the resulting pseudo-table T to its corresponding content tree CT. Since T contains the caption C



A single dot (.) is the dot operator as presented in Definition 3.

Figure 7: Mapping from a column-row-wise HTML table to a pseudo-table

A test table with merged cells

	Average.height	*Average.weight*	*Red eyes*
Males	1.9	0.003	40%
Females	1.7	0.002	43%

Figure 9: The pseudo-table of the HTML table in Figure 1(b)

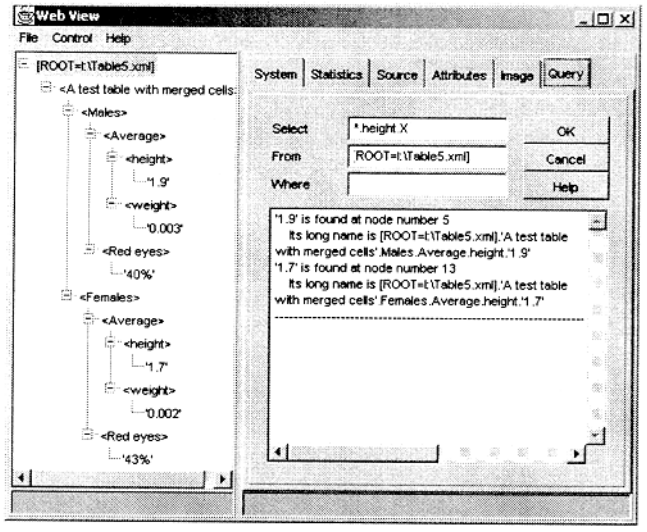


Figure 10: CT of the HTML table in Figure 1, rendered in WebView, and a sample query

(i.e., *'A test table with merged cells'*), C forms the root node of the resulting CT according to Definition 5. Next, consider C_1. Since C_1 is empty, we skip C_1 in the hierarchy $V_R \leftarrow C_1 \leftarrow a_{i,1} \leftarrow \ldots$ and create two child nodes of the root node C by using $a_{1,1}$ (i.e., `Males`) and $a_{2,1}$ (i.e., `Females`). The rest of the cells $a_{i,j}$ $(1 \leq i \leq 2,\ 2 \leq j \leq 4)$ in the last two rows of T yield nodes and edges in CT as follows: $a_{1,1} \leftarrow C_2 \leftarrow a_{1,2}$; $a_{1,1} \leftarrow C_3 \leftarrow a_{1,3}$; $a_{1,1} \leftarrow C_4 \leftarrow a_{1,4}$; $a_{2,1} \leftarrow C_2 \leftarrow a_{2,2}$; $a_{2,1} \leftarrow C_3 \leftarrow a_{2,3}$; $a_{2,1} \leftarrow C_4 \leftarrow a_{2,4}$. We render the resulting CT in WebView [LN99], as shown under the root node "[ROOT=E:\Table5.xml]" in the left pane of Figure 10, where each node label is enclosed within angle brackets except the leaf nodes. Note that the association of a table data D with another, i.e., the intended hierarchy of D in the table, can be conceived by examining the long name of D. For instance, `'A test table with merged cells'.Males.Average.height.'1.9'` captures the intended hierarchy of "1.9". Also, note that CT does not contain any HTML tags, and hence CT has less number of nodes than the corresponding syntactic hierarchy as shown in Figure 2. (The content tree of the table in Figure 4(b) is as shown in the left pane of Figure 11.) □

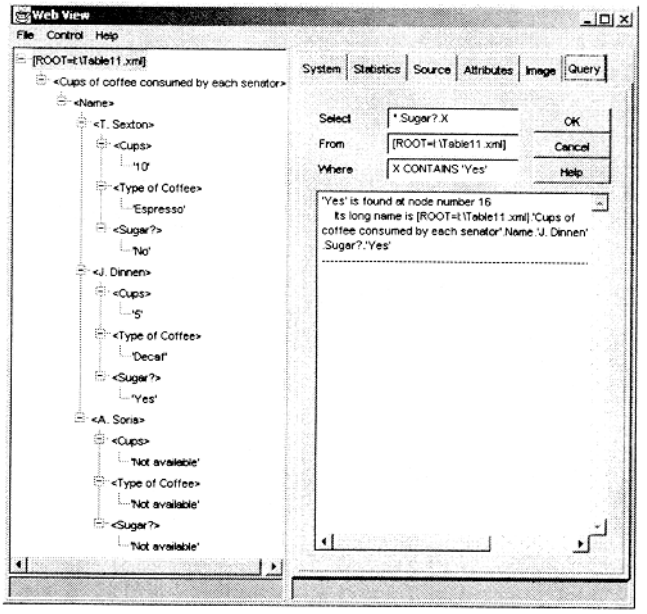


Figure 11: CT of the HTML table in Figure 4(b), rendered in WebView, and a sample query

3.3 Complexity of the content tree approach

Our approach on constructing the content tree of a given HTML table is a "one-pass" approach, meaning that the approach processes the given source table in the left-to-right, top-to-bottom, object-by-object manner, and never reads the same object in the given table more than once. Upon detecting an object in the source table, our approach takes appropriate actions and process the next object until no more object is found in the source table. Furthermore, no preprocessing is required. Hence, the time complexity of the proposed approach is proportional to the number of objects in the given table.

3.4 Implementation of the approach

The proposed approach for retrieving hierarchical data from HTML tables has been implemented as a Java class and tested on a Pentium-based workstation using the JDK 1.1.7. The Java class generates the content tree of a given HTML table T in the lexical form whose representation is similar to the lexical SDT of T. For rendering the content tree graphically, WebView [LN99] can be used (see the left pane of Figures 10 and 11). Since the proposed approach for constructing the content tree of an HTML table is implemented as a Java class, which is portable across different platforms, it is easy to integrate our content-tree construction approach

into an existing data acquisition tool, including wrappers for data warehousing systems.

For some applications, it might be appropriate to apply the proposed approach to HTML tables that are dynamically generated via CGI or others. In such a case, a front-end processor can easily be plugged into our approach to fetch those dynamically generated tables. A demonstration of fetching tables dynamically from an existing Web site can be found at the beginning of Section 4.3.

4 Applications of content trees

4.1 Hierarchy-based queries

We have implemented a simple SQL-like query processor in WebView that takes the advantages of data hierarchies in HTML documents. (The detailed discussion on the syntax and semantics of the WebView query language can be found in [LN99].) A WebView query statement consists of a *SELECT* clause followed by a *FROM* clause, with an optional *WHERE* clause at the end of the query statement. An attribute specified in a SELECT statement can be in the form of a long name or path expression mixed with variables and wild cards, which are special variables. In the FROM clause, we specify a (subtree of a) content tree as the search domain for the desired data specified in the SELECT clause.

The query tab on the right pane of Figure 10 shows a simple query which retrieves the value of any `height` in the content tree (as shown on the left pane) which is generated from the HTML table in Figure 1 and is specified in the *FROM* clause. Since there are two nodes in the content tree whose names are `height`, both 1.9 and 1.7 are retrieved, and their long names are returned by the WebView query processor. Furthermore, Figure 11 illustrates how to query senators whose *Sugar?* field has the value '*Yes*' (using the WHERE clause) from the HTML table in Figure 4(b) whose content is captured in its respective content tree which is displayed on the left pane of Figure 11. The long name of the retrieved data of this query contains "J. Dinnen."

4.2 HTML to XML

As shown in Figures 10 and 11, the content tree of an HTML table captures the hierarchical relationships of the data, not HTML tags, contained in the HTML table. Using the resultant content tree T of an HTML table H, we can easily convert H into an XML document. Consider the data hierarchy, besides the root node, of T and the following algorithm:

Algorithm: HtmlTable2Xml
Input: The content tree T of an HTML table H
Output: XML document X of H

1. Initialize the resulting XML document X by appending the XML declaration `<?xml version="1.0"?>` to X. /* This declaration is generic enough for any XML documents. */
2. Append `<!DOCTYPE root SYSTEM "extern.dtd">`, a placeholder, for a document type declaration to X. /* Since at this stage we are unable to determine a DTD of X, we append a placeholder to X such that `root` and `extern` will be replaced later by a real identity when they become available. */
3. Traverse the content tree T from top to bottom and from left to right. For each node v in T, DO:
   (a) Replace string _s in the name of v by U and store the resulting v to e, where '_' is a sequence of one or more spaces, s is an alphanumeric letter, and U is the upper-cased s. This replacement is necessary since no space is allowed in the name of an XML element.
   (b) Attach[3] e to X while maintaining the hierarchy of e to its immediate container element.
   (c) Update a DTD[4] of X.
4. Replace `root` and `extern` in the document type declaration by the name of the first XML element in X.

Example 4 Consider the content tree T as shown in the left pane of Figure 10 and algorithm HtmlTable2Xml. After steps 1 and 2, X contains `<?xml version="1.0"?> <!DOC TYPE root SYSTEM "extern.dtd">`. At step 3(a), the first node `<A test table with merged cells>` in T yields `<A TestTableWithMergedCells>`, which is appended to X by step 3(b). Note that `</ATestTableWith MergedCells>` will be appended to X after all other nodes in T are processed according to the definition of *attach*. The next node `<Males>` in T is appended to X immediately after `<ATestTableWith MergedCells>` as an immediate content. Also, `<Average>`, `<height>`, and 1.9 are appended to X in their respective order. Hereafter, `</height>` is appended and `<weight>` is processed. Eventually, `</ATestTableWithMergedCells>` is appended to X. By step 4 `root` and `extern` in the document type declaration are replaced by ATestTableWithMergedCells. The resultant X is shown below:

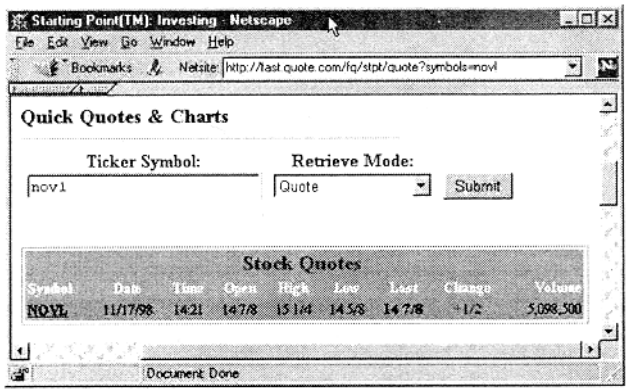
```
<?xml version="1.0"?>
<!DOCTYPE ATestTableWithMergedCells SYSTEM
 "ATestTableWithMergedCells.dtd">
<ATestTableWithMergedCells>
 <Males>
  <Average>
   <height> 1.9 </height> <weight> 0.003 </weight>
  </Average>
  <RedEyes> 40% </RedEyes>
 </Males>
 <Females>
  <Average>
   <height> 1.7 </height> <weight> 0.002 </weight>
  </Average>
  <RedEyes> 43% </RedEyes>
 </Females>
</ATestTableWithMergedCells> □
```

4.3 Data warehousing

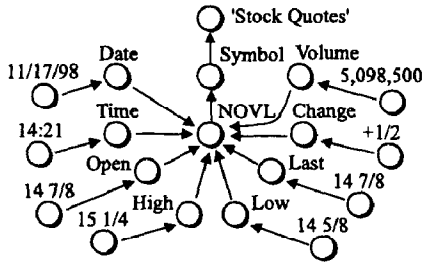
In this section, we demonstrate the construction of repositories for a data warehousing system by using the content trees of HTML tables, which are the source of the repositories. To demonstrate repository construction by using content trees, we consider a stock quotations service available at Starting Point(TM) through the URL *http://fast.quote.com/fq/stpt/quote?symbols* = X, where X is a valid ticker symbol. Given a valid ticker symbol, the service provides the open, high, low, and last values, the change of the value, and the volume of the requested ticker symbol at the requested time and date. The requested information is returned to the browser as an HTML table. A sample resulting table with X = *novl* is shown in Figure 12(a), and the content tree

[3] The operation *attach*(e) appends e to X as a start-tag when the program thread enters in the scope of e and appends the end-tag of e when the process exits from the scope of e.

[4] A detailed discussion on how to construct a DTD of an XML document is excluded since the discussion is beyond the scope of this paper.



(a) Returned quotations in an HTML table



(b) The content tree generated for Figure 12(a)

Figure 12: Stock quotations service at Starting Point(TM)

of the resulting table is shown in Figure 12(b). Recall that a data component in a content tree can be accessed by its long name. For instance, the value of *change* (*in price*) of *NOVL* on November 17, 1998 can be expressed as `'Stock Quotes'.Symbol.NOVL.Change.+1/2`.

To demonstrate repository construction, we construct two types of repositories, a *semistructured repository*, which is based on the semistructured data model, and a *relational repository*, which is based on the relational data model. By constructing these two repositories, we demonstrate what types of materialized views of the incoming data are maintained and served by the data warehousing system discussed in this section.

In our data warehousing model, data acquisition from the source is handled by the wrapper of the warehousing system, and the proposed content-tree construction method is integrated into the wrapper. We assume that the engine for fetching the source HTML tables has already been built in the wrapper (see WebView [LN99]). The acquired data, in the form of a content tree, is passed to two integrators, which are responsible for maintaining the *semistructured view* and the *relational view* of the data, respectively. In the design of our sample warehousing system, the primary role of the wrapper is to provide a uniform representation of data for any ticker symbol on any particular date to each integrator. The role of an integrator is to maintain a uniform view of the data, which is either a semistructured data or a relational table.

Note that the stock quotations service provides updated information for a given ticker symbol every minute. Hence, the wrapper can be fired as often as necessary depending on the demand on the warehouse. We assume that storing stock quotations on a daily basis is sufficient for our need in this demonstration. By firing the wrapper for each ticker symbol on a daily basis, we keep a daily snapshot of the quotations and maintain the history of the stock price for every ticker symbol. Note that a *symbol*, which represents a ticker symbol, is required in a query in order to obtain the desired stock quotation. A ticker symbol is specified in the query string[5] as part of an URL. For example, *symbols* = *novl* specifies the ticker symbol *NOVL*, and the wrapper is capable of acquiring stock quotation for any ticker symbol by replacing *novl* with the respective ticker symbol.

4.3.1 Semistructured Repository

Figure 13 illustrates the data flow from the Internet data source, which includes the stock quotations service in our example, to the two repositories of the data warehouse. As shown in the figure, the content tree of the fetched table from the data source represents a snapshot of the quoted stock price for a particular ticker symbol on a particular date. As an example, since a content tree that is passed from the wrapper contains the data of a particular ticker symbol on a particular date and time, the semistructured integrator makes an alteration on the content tree in order to maintain the view of the data according to the order of ticker symbols and dates. The alteration is made by adjusting the dependency of the data objects in the tree so that the value of *Symbol* (i.e., *novl*) becomes the root node of the tree and *Date* is attached to the root node. Finally, this altered content tree is attached to the root node of the repository (i.e., *'Stock Quotes'*). With that the integrator is capable of providing a uniform view of the data that are searchable by using a ticker symbol and a date.

As illustrated in Figure 13, the long name of a data component is altered by the semistructured integrator in the repository. For instance, the value of *change* (*in price*) of *NOVL* on November 17, 1998 is expressed as `'Stock Quotes'.NOVL.Date.11/17/98.Change.+1/2`. Using a semistructured-data query language [AQM+97, LN99], a query "What is the change in price of *NOVL* on November 17, 1998?" can be posted against the semistructured repository as `SELECT` *_.'Change'.X* `FROM` *'Stock Quotes'.'novl'.'Date'.'11/17/98'*, where strings surrounded by single quotes are constants, X is a variable, and '_' is the anonymous variable. The `FROM` clause returns the subtree S rooted at `'Stock Quotes'.'novl'.'Date'.'11/17/98'` from the data stored in the semistructured repository. The `SELECT` clause binds the name of the child of *Change* to variable X. According to the condition specified in this `SELECT` clause, the *Change* node is a child of the node `'11/17/98'` in S. The answer to the query is *+1/2*, since the anonymous variable in the `SELECT` statement is bound to the root node of S, which is `'11/17/98'`.

4.3.2 Relational repository

The architecture described in the previous section can easily be adopted for building a relational repository since the proposed content-tree construction method is capable of transforming an HTML table to a pseudo-table, as discussed in Section 3.2.

Data acquisition for the relational repository can be handled by the same wrapper for the semistructured repository. For the relational repository, the integrator integrates the

[5] A query string is the string following the first question mark ("?") in an URL.

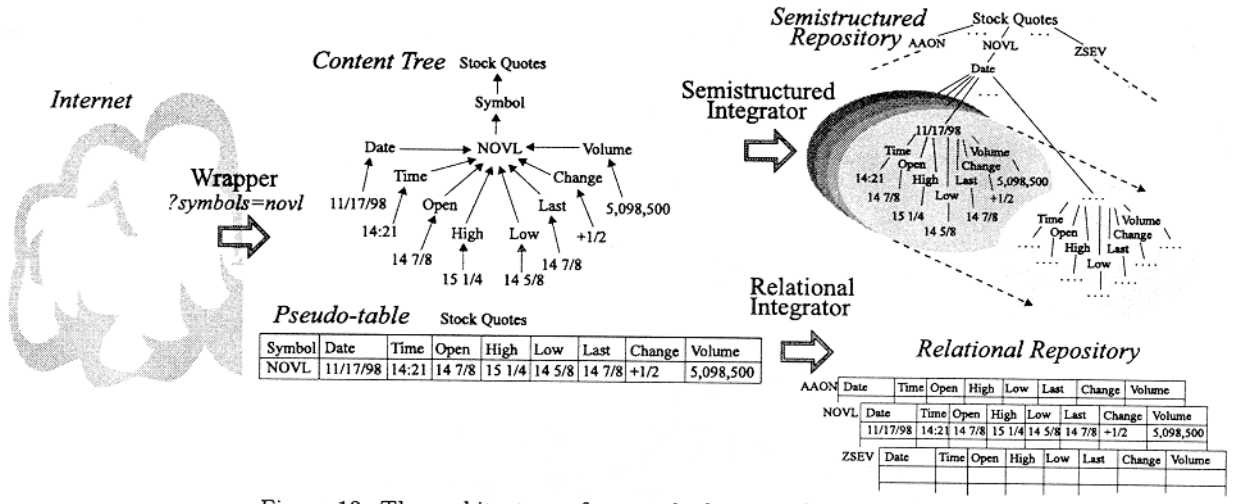


Figure 13: The architecture of a sample data warehousing system

acquired data into relational tables by altering the pseudo-tables passed from the wrapper in order to maintain and provide a uniform, relational view of the data for any ticker symbol. In our sample warehousing system, we create a separate table for each ticker symbol such that the relation scheme of the table is of the form $R(Date, Time, Open, High, Low, Last, Change, Volume)$, where R is the corresponding ticker symbol and *Date* is the key attribute of R. For an instance D of the pseudo-table for a particular ticker symbol on a particular date, the relational integrator first locates the destination table R such that R matches the content of the *Symbol* cell of D and then inserts into R a new record out of the data cells of D accordingly, with the exception of the heading rows and the *Symbol* cell. The query, "What is the change in price of *NOVL* on November 17, 1998?", can be posted against the repository as `SELECT` *Change* `FROM` *NOVL* `WHERE` *Date* = 11/17/98.

5 Concluding remarks

HTML tables become widely used in Web documents to present data in a tabular structure. The content tree, as well as its corresponding pseudo-table, construction approach presented in this paper can construct the content tree of the data contents in a given HTML table without requiring the internal structure of the table to be known beforehand. The content tree captures the intended hierarchy of the data contents, i.e., the association of each data content with others, in the table, in contrast to the syntactic hierarchy which captures the hierarchy of tags as well as the data contents in the table based on the HTML grammar. A content tree is also more efficient than its corresponding syntactic tree in terms of the cost of traversing the tree and space for storing the tree. It suits better than its corresponding syntactic tree for querying hierarchically structured table data since the hierarchy of the objects involved in a query can be asserted in a content tree without using tags. The time complexity of the proposed content-tree construction approach is proportional to the number of tags and data in the given table, and our approach can be used by existing wrappers and integrators (in data warehousing systems), which frequently access a large number of HTML tables for extracting hierarchically structured data, such as stock quotations, medical data, etc., to automate the data acquisition process. We also discover the advantages of using content trees in converting their corresponding HTML tables to XML documents.

References

[ACC+97] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Simeon. Querying Documents in Object Databases. *Journal on Digital Libraries*, 1(1):5–19, April 1997.

[AM98] G.O. Arocena and A.O. Medelzon. WebOQL: Restructuring Documents, Databases and Webs. In *Proceedings of the 14th Intl. Conf. on Data Eengineering*, February 1998.

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 206–215, August 1997.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *Journal on Digital Libraries*, 1(1):68–88, 1997.

[HGMC+97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting Semistructured Information from the Web. In *Proceedings of the Workshop on Management of Semistructured Data*, May 1997.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A Query System for the World-Wide Web. In *Proceedings of the 21st Intl. Conf. on Very Large Data Bases*, pages 54–65, Sept. 1995.

[LN98] S.-J. Lim and Y.-K. Ng. Constructing Hierarchical Structures of Sub-Page Level HTML Documents. In *Proceedings of the 5th International Conference on Foundations of Data Organizations*, pages 66–75, November 1998.

[LN99] S.-J. Lim and Y.-K. Ng. WebView: A Tool for Retrieving Internal Structures and Extracting Information from HTML Documents. In *Proc. of the 6th Intl. Conf. on Database Systems for Advanced Applications*, pages 71–80, 1999.

[MM97] A. Mendelzon and T. Milo. Formal Models of Web Queries. In *Proceedings of the 16th International Symposium on Principles of Database Systems*, pages 134–143, 1997.

[RHJ98] D. Raggett, A. Hors, and I. Jacobs. HTML 4.0 Specification - W3C Recommendation. http://www.w3.org/TR/REC-html40, 1998.