RECORD-BOUNDARY DISCOVERY IN WEB DOCUMENTS

by

Yuan Jiang

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

December 1998

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Yuan Jiang

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Dennis Yiu-Kai Ng, Chair
Date	David W. Embley
Date	Dan R. Olsen

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Yuan Jiang in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Dennis Yiu-Kai Ng
Chair, Graduate Committee

Accepted for the Department

Scott N. Woodfield
Graduate Coordinator

Accepted for the College

Nolan F. Mangelson, Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

RECORD-BOUNDARY DISCOVERY IN WEB DOCUMENTS

Yuan Jiang

Department of Computer Science

Master of Science

Extraction of information from unstructured or semistructured Web documents often requires a recognition and delimitation of records. (By "record" we mean a group of information relevant to some entity.) Without first chunking documents that contain multiple records according to record boundaries, extraction of record information will not likely succeed. In this thesis we describe a heuristic approach to discovering record boundaries in Web documents. In our approach, we capture the structure of a document as a tree of nested HTML tags, locate the subtree containing the records of interest, identify candidate separator tags within the subtree using five independent heuristics, and select a consensus separator tag based on a combined heuristic. Our approach is fast (runs linearly for practical cases within the context of the larger data-extraction problem) and accurate (100% in the experiments we conducted).

ACKNOWLEDGMENTS

I would like to express my deep appreciation to several people for their support and assistance during my MS thesis work. Foremost, I am grateful to my mentor, Dr. Dennis Ng, who has always made himself available to discuss ideas and provide feedback. We have spent many hours together working over ideas and finding ways to improve our approach. Without his guidance and enthusiasm, this work would not have been finished on time and the quality of the thesis would have been much lower.

I appreciate the support of Dr. David Embley, who is my second committee member and the co-director of the Data Extraction research group. His insights and enthusiasm really helped me to accomplish this work. Those weekly meetings were very helpful in shaping many of the ideas that comprise my research. I appreciate his time and effort on my behalf.

The entire Data Extraction research group has also been helpful to me. I would like to acknowledge Dr. Stephen Liddle who provided some codes and figures for my ontology- matching heuristic and Dr. Dallan Quass who assisted me in selecting and categorizing the related research work from the others. I would also like to acknowledge the valuable suggestions of the other members of the group.

I want to express my appreciation to Dr. Dan Olsen for stepping in at the last minute with only three weeks notice when a previous committee member was unable to attend my thesis defense.

Last, but by no means least, I would like to thank my family and many friends for their support and encouragement while I attended school.

Contents

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

2 Using Tag Trees for Locating Records of Interest **7**

2.1 The Structure of Tag-trees 9

2.2 Algorithms to Construct Tag Trees 12

2.3 The Largest Fan-out Heuristic Approach 14

2.4 The Pre-processing of the Tags 19

3 The Individual Heuristics **22**

3.1 The Ontology-Matching Heuristic 23

3.2 The Repeating-Tag Pattern Heuristic 33

3.3 The Standard Deviation Heuristic 34

3.4 The Identifiable "Separator" Tags Heuristic 39

3.5 The Highest-Count Tags Heuristic 44

4 The Combined Heuristics **46**

4.1 Certainty Measure 47

4.2 Initial Experiments 47

4.3 The Combined Heuristic 51

4.4 The Record-Boundary Discovery Algorithm 53

5 Experimental Results **57**

6 Concluding Remarks **62**

List of Tables

3.1 The record separators and the numbers of their appearances in 100 Web documents 42

4.1 On-line newspapers chosen for initial experiments 49

4.2 Experimental results for obituaries application 49

4.3 Experimental results for car advertisements application 49

4.4 Certainty factors, as selected by our initial experiments 50

4.5 Experimental results for all the combined heuristics 52

5.1 Test set 1 - obituaries 60

5.2 Test set 2 - car advertisements 60

5.3 Test set 3 - computer job advertisements 60

5.4 Test set 4 - university course descriptions 60

5.5 Success rates of individual heuristics and ORSIH for experimental Web documents 61

List of Figures

1.1 Data extraction and structuring process 5

2.1 A sample Web document 11

2.2 A node that includes the hypertext in Line 4 of the Web document in Figure 2.1 12

2.3 The tag tree of the Web document in Figure 2.1 13

2.4 A sample Web document after all the "missing" end-tags have been inserted 17

2.5 The tag tree of the Web document in Figure 2.1 with the largest-fan-out sub-tree rooted at node *td* which is embedded within an ellipse . 19

3.1 A sample ontology in graphical form 25

3.2 The ontology rules for the obituaries application 27

3.3 The Data-Record Table generated by using the ontology rules in Figure 3.2 30

4.1 Graphical car advertisements ontology 50

5.1 Graphical computer job advertisements ontology 58

5.2 Graphical university course descriptions ontology 58

5.3 The ontology rules for the OM heuristic 59

0.5in

Chapter 1

Introduction

The amount of data available electronically on the Web has increased dramatically in recent years. At present, browsing and keyword searching are the two commonly used methods for retrieving data from the Web; however, these methods are ineffective and have severe limitations [Ape94]. Browsing the Web is inefficient, as users have to read the documents in order to locate the desired information. In addition, browsing is not suitable for locating particular data items because it is easy to get lost while tracing links on the Web. Keyword searching, on the other hand, often returns a huge amount of data, so much that a user cannot handle it.

To overcome these limitations, some researchers have resorted to database techniques, which require structured data. It has been realized that most of the Web data are semistructured in nature [Abi97, BDFS97], which means that they do not conform to a regular nor rigorous structure like data in a relational database. Furthermore, their schema may change dynamically. Even though relational database query languages are well-developed, widely accepted, and easy to use, we cannot adopt these languages for retrieving data from the Web since Web documents are not structured as relational databases. In order to query Web documents using traditional database query languages, attempts have been made to build wrappers around documents [AK97a, AK97b, AM97, DEW97, GHR97, HGMC$^+$97, KWD97, Sod97, Ade98,

ECJ+98, MMK98]. A *wrapper* is a procedure that extracts data in Web documents and structures them into some regular forms (such as relational database tables).

In building wrappers, we often need to divide source Web documents into chunks of information that correspond to records. Each of these records is similar in structure to other records in the same Web document and represents information relevant to some entity. For example, in a Web document that contains a list of car advertisements, a record in the document is a single car advertisement. Identifying the structure of records of interest in a Web document is the first step in designing a wrapper for extracting data from the document. Since different Web documents have different structures, the record identification task, by itself, is nontrivial [AK97a, AK97b].

In this thesis we propose a heuristic approach to discover boundaries of records in a Web document that contains multiple records. Once found we can separate these records and pass them on for further processing, i.e., extracting data from the records. The main contribution of this thesis is to provide a set of individual heuristics and a way to combine these heuristics into a method for discovering record boundaries. We focus on Web documents that are written in HTML[1] and assume that each Web document we process (1) has multiple records of interest and (2) contains at least one record-separator tag.

This is not the first time the problem of separating records in a Web document has been addressed. [AM97, HGMC+97] detect record boundaries manually. They first examine the documents, find the HTML tags that separate the records of interest, and then write a program to separate the records. [AK97a, AK97b, DEW97, KWD97, Sod97, Ade98] separate records with some degree of automation. Their approaches focus primarily on using syntactic clues, such as HTML tags, to iden-

[1]Even though we have done all our work with HTML documents, most of this work should carry over directly to other document type definitions, such as XML.

tify record boundaries and then separate the records. Some of these approaches [AK97a, AK97b, Ade98] require user assistance in locating record boundaries. None of these approaches is fully automatic.

Our approach differs markedly from these proposals. We present our approach and results as follows. We first provide a heuristic for locating groups of records within a Web document D (in Chapter 2). D, which contains the records of interest, usually contains other irrelevant information such as a header and a trailer. To eliminate irrelevant information from the consideration and concentrate on the region of D that contains the records, this heuristic requires the construction of the tag tree of D based on the nested structure of start- and end-tags and locates the sub-tree that contains the records of interest in D. We restrict our search for a record-separator tag to candidate tags, tags that are considered as record-separator tags, found in this sub-tree. After locating the sub-tree, we apply five different heuristics, each of which individually attempts to locate a record-separator tag among the candidate tags that appear in the sub-tree (in Chapter 3). These heuristics are ontology-matching (OM), repeating-tag pattern (RP), standard deviation (SD), identifiable "separator" tags (IT), and highest-count tags (HT). Each of these heuristics returns one or more candidate separator tags with a measure of certainty/uncertainty attached to each candidate. Finally, we adopt Stanford certainty theory [LS98] to combine these individual heuristics to determine a consensus record-separator tag (in Chapter 4). For practical cases and in the context of our overall data-extraction process, the entire record-boundary discovery process is $\mathcal{O}(n)$, where n is the size of a given Web document. We applied this approach in four different application areas using Web documents obtained from twenty different sites, which together contained thousands of records (in Chapter 5). The results were uniformly good, attaining 100% accuracy on all sites we examined.

Before explaining the details of our approach, we begin with a short description of the larger context in which we use our record-boundary-discovery heuristics. This short description is necessary to provide the context for our record-boundary-discovery research work (which is a portion of the on-going data-extraction project [WWW]) and to explain what we mean by an ontology and how to use it in our work.

Figure 1.1, which is taken from [ECJ+98], shows the overall process for extracting and structuring Web data. As depicted in the figure, the input (upper left) is a Web page, and the output (lower right) is a populated database. The figure also shows that an application ontology is an independent input. For us, an application ontology is a conceptual model augmented with additional information to describe constants and keywords for the application. This ontology describes the application of interest. When we change applications, for example from car advertisements, to job advertisements, to obituaries, to university course descriptions, we only change the ontology, and apply the same process to different Web pages. Significantly, everything else remains the same: the routines that extract records, parse the ontology, recognize constants and keywords, and generate the populated database instance do not change. In this way, the overall process is generally applicable to any application domain.

Specifically, the overall data-extraction project consists of the following steps. (1) We develop the ontological model instance for the domain of interest (the *Application Ontology* in the figure). (2) We parse this ontology to generate a database scheme (the *Database Description* in the figure) and to generate rules for matching constants and keywords (the *Constant/Keyword Matching Rules* in the figure). (3) To obtain data from the Web, we invoke a *Record Extractor* (see figure) that separates an unstructured Web document into individual record-size chunks, cleans them by removing markup-language tags, and presents them as individual unstructured documents for further processing. (It is the record separation task in this component that we discuss

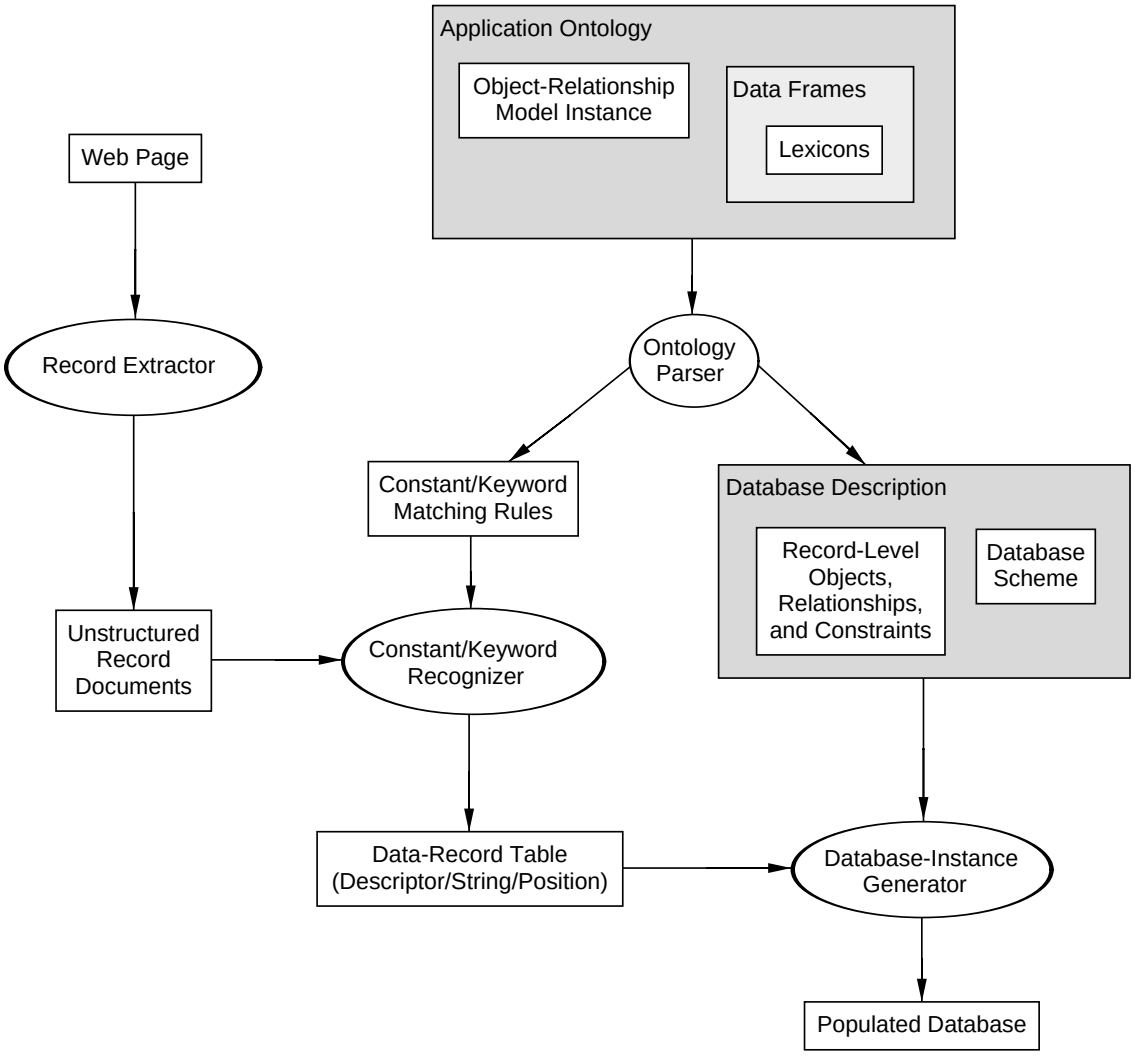

Figure 1.1: Data extraction and structuring process

in this thesis.) (4) We invoke recognizers that use the matching rules generated by the parser to extract from the cleaned individual unstructured documents the objects and relationships from which we obtain the raw, as-yet-unorganized data to populate the model instance. The result is the *Data-Record Table* in the figure. (5) Finally, we populate the generated database scheme by using heuristics to determine which constants populate which records of the database scheme. These heuristics correlate extracted keywords with extracted constants and use cardinality constraints in the ontology to determine how to construct records and insert them into the database.

Chapter 2

Using Tag Trees for Locating Records of Interest

Most Web documents are hypertext documents that are written according to a *Document Type Definition (DTD)*[1] and include plain text and tags. A *tag* in a Web document consists of a pair of opening and closing brackets (i.e, "<" and ">") that enclose a tag name, sometimes followed by a list of tag attributes, whereas *plain text* in a Web document is a sequence of characters not embedded within any tag. We distinguish each tag in a Web document as either a start-tag or an end-tag. A *start-tag* is a tag whose name does not start with a forward slash (i.e., "/"), whereas the name of an *end-tag* is the name of its corresponding start-tag preceded by a "/". A start-tag may not have a corresponding end-tag. In this thesis, we discard and thus totally ignore two types of tags: (1) comment tags that start with <! and (2) any end-tag that has no corresponding start-tag. We categorize these two types of tags as *useless tag*s.

Tags in a Web document D define discrete regions in D. A *region* R in D begins where a start-tag S appears and ends where the corresponding end-tag of S appears. If the corresponding end-tag of a start-tag S does not exist, we consider whatever

[1] A *DTD* defines the syntax and semantics of a language used for creating a Web document. For example, every HTML document should conform to the HTML SGML DTD, the Document Type Definition that defines the HTML standard.

appears between S and the next tag (or the end of D) as a region. Between a start-tag and its corresponding end-tag, other start- and/or end-tags can be nested. Note that regions, as defined here, do not necessarily correspond to regions over which a tag applies for display purposes. Our purpose here is not to display a document, but to build a convenient structure for discovering record boundaries.

Based on this nested structure, we construct a data structure, called a tag tree, to represent a document D according to the regions in D. The *tag tree* of D is a tree representation of D, which is based on the tags and the plain text that appears either between each pair of start-tag and its corresponding end-tag in D or after the end-tag, if it exists. A node in the tag tree of D identifies a region in D.

We attempt to discover the boundaries of records in D by using the tag tree T of D and an automated tool - the record-boundary discovery program. In designing our record-boundary discovery program, we first attempt to detect the region in T that most likely contains the records of interest. It is our conjecture that in a Web document with multiple records of interest, the sub-tree of T rooted at the largest-fan-out node N, called the *largest-fan-out sub-tree* of T, should contain the records. The *largest-fan-out node* in the tag tree T has the largest number of child nodes among all the other nodes in T. We count the number of appearance of each distinct start-tag in the region identified by N, and distinguish each of these tags as either an irrelevant tag or a candidate tag. An *irrelevant tag* is a start-tag with relatively few appearances among the start-tags appeared in the region identified by N, whereas a *candidate tag* is a start-tag that appears significantly more often than the irrelevant tags in the region bound by N. We ignore all the irrelevant tags and consider only the candidate tags in all the heuristics to be introduced in Chapters 3 and 4. Eliminating all the irrelevant tags from consideration simplifies our task in determining the correct *record separator* (which is a start-tag that separates two records of interest) of D. If

there is only one candidate tag, we treat it as the record separator; otherwise, we apply the heuristic approaches to determine the record separator.

We introduce the structure of tag trees in Section 2.1 and the algorithms to construct a tag tree in Section 2.2. In Sections 2.3 and 2.4 we describe the process of finding the largest-fan-out sub-tree S of a tag tree and eliminating from consideration all the irrelevant tags in the region bound by S.

2.1 The Structure of Tag-trees

The structure of a node in the tag tree T of a Web document D is shown below:

Start-tag
Inside-text
Post-text

A node N in T consists of three elements, the *start-tag* S, the *inside-text* I, and the *post-text* P. The content of any of these three elements may be unassigned. The content of S, denoted $N.S$, is a start-tag S_t. $N.S$ is not assigned only if N is the root node of T, which represents the region that is the entire D. The content of I, denoted $N.I$, is the plain text that appears between S_t and the next tag (or the end of D). If the next tag is the corresponding end-tag E_t of S_t, then N is a leaf node of T. The content of P, denoted $N.P$, is the plain text that appears between E_t and the end of D if E_t is the last tag in D, or it is the plain text that appears between E_t and the next tag in D.

Figure 2.1 shows a sample Web document[2]. The numbering of each line is not part of the document but is included solely for referencing a particular portion of the

[2]To protect individual privacy, this document is not real. It is based on an actual Web document, but it has been significantly changed so as not to reveal the identities of individuals. However, Web documents used in our experiments reported in Chapters 4 and 5 are real.

document. Figure 2.2 shows a node in the tag tree of the Web document in Figure 2.1. In Line 4 of the document, the plain text "Funeral Notices - " appears between the start-tag <h1 align="left"> and its corresponding end-tag </h1>. Thus, the content of the inside-text element of this node is "Funeral Notices - ". Since the plain text "October 1, 1998" appears between </h1> and the next tag <hr>, the content of the post-text element of this node is "October 1, 1998".

If there exist other tags between a start-tag S_t and its corresponding end-tag E_t of the region denoted by node N in document D, then new nodes, called *child nodes* of N, are constructed while tag tree T is being built. N represents a region R of D, whereas a *sub-region* of R, which is denoted by a child node of N, is a region that starts with a start-tag S' and ends with its corresponding end-tag E' (or the next tag if E' does not exist), and S' and E' are not embedded within other start-tags and their corresponding end-tags other than S_t and E_t. If there are m sub-regions in R, then there are m child nodes of N. The order of these child nodes in T is determined by the appearance of their corresponding start-tags in R. We apply the same strategy of constructing N to construct its child nodes. The leftmost child node of N, which represents the first sub-region of R, contains the hypertext bound by the first start-tag B_s after S_t in R and its corresponding end-tag B_e (or the next tag if B_e does not exist). The second child node of N represents the second sub-region of R, and so on. If there exist other start-tags embedded inside one of the sub-regions in R, we apply the same method for constructing the next generation of the child nodes at the next level. Edges from a parent node to its child nodes capture the nested structure of tags.

Figure 2.3 shows an abstraction of the tag tree T of the Web document D in Figure 2.1. Note that in the abstraction we use only the name in the start-tag element of a node in T as the label of the node to simplify the drawing of each node in the

```
1: <html><head><title>Classifieds</title></head>
2: <body bgcolor="#FFFFFF">
3: <table><tr><td>
4: <h1 align="left">Funeral Notices - </h1> October 1, 1998
5: <hr>
6: <b>Lemar K. Adamson</b><br> age 84, of Tucson, died September 30, 1998.
7: He is survived by wife, Cindy; daughters, Elvia, Gloria, Irene, Isabel,
8: Jewel, and Jessica; sons, Paul, John, Jeffery, and Louis; brothers, Kirk,
9: Justin, Ivan, Hubert and Grover. Funeral service 10:00 a.m. Monday,
10: October 5, 1998 at Silverbell Ward, 1540 E. Linden. Burial in City
11: Cemetery. Friends may call from 9:00 a.m. to 10:00 a.m. Monday, at the
12: church. Arrangements by <b>MEMORIAL CHAPEL</b>, 236 S. Scott<br>
13: <hr>
14: Our beloved <b>Brian Fielding Frost</b>, age 41, passed away Wednesday
15: morning, September 30, 1998, due to injuries sustained in an automobile
16: accident. He was born January 12, 1957 in Salt Lake City, to Donald
17: Fielding and Helen Glade Frost. He married Susan Fox on June 1, 1981.
18: He is survived by Susan; sons Alfred, Joseph; parents, and two sisters,
19: Anne (Dale) Elkins and Sally (Kent) Britton. Funeral services will be
20: held at 12 noon Tuesday, October 6, 1998 in the <b>Howard Stake Center</b>,
21: 350 South 1600 East. Friends may call 5-7pm. Monday at
22: <b>Carrillo's Tucson Mortuary</b>, 3401 S. Highland Drive. Interment at
23: Holy Hope Cemetery.<br>
24: <hr>
25: <b>Leonard Kenneth Gunther</b><br> age 82. A resident of Tucson,
26: passed away peacefully on September 30, 1998. He was born June 6, 1916 in Iowa.
27: He joined the U.S. Navy serving during World War II. He remained a member
28: of the U.S. Naval Reserve (USNR) for several years. He is survived by his wife,
29: Gwendolyn; sons, Eric D. of San Francisco, CA, Vincent J. of Tucson; a daughter,
30: Janet H. of Provo, UT; and one granddaughter, Sarah R. of Phoenix, AZ.
31: Friends may call from 5:00 p.m. until 7:00 p.m. on Monday, October 5, 1998 at
32: <b>HEATHER MORTUARY</b>, 1040 N. Columbus Blvd. Funeral services
33: will be at 11:00 a.m. at <b>HEATHER MORTUARY</b>, on Tuesday,
34: October 6, 1998. Burial will be private at South Lawn Cemetery.<br>
35: <hr>
36: </td></tr></table>
37: All material is copyrighted.
38: </body>
39:</html>
```

Figure 2.1: A sample Web document

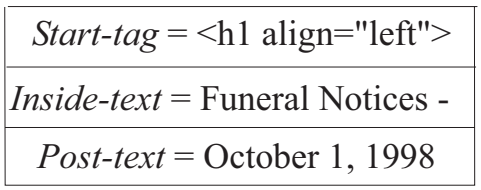

Figure 2.2: A node that includes the hypertext in Line 4 of the Web document in Figure 2.1

figure. Since the tags <html> and </html> embed all of D, <html> is the content of the start-tag element of the root node R of T. Since a <head> tag exists between the <html> and </html> tags in D, a child node of R is constructed which contains <head> as the content of its start-tag element. The contents of the inside-text and post-text elements of node *head* are unassigned. This is because there is no plain text between <head> and the next tag <title>, and between the </head> tag and the next tag <body>. The tag <title> is embedded between the <head> and </head> tags in D; it is thus a child (and the only child) of node *head*. The content of the inside-text element of node *title* is "Classifieds," whereas the content of its post-text element is unassigned. The node labeled *body* is another child node of R, and the descendant nodes of the node *body* are constructed as shown in Figure 2.3.

2.2 Algorithms to Construct Tag Trees

The tag tree of a Web document is constructed by algorithm *Construct_Tree* presented in this section. The algorithm, which takes a Web document D as input and produces the tag tree T of D as output, consists of three steps. (1) A stack S and a table TBL, whose entries are to be indexed by the start-tag names in D, are initialized. S is used for keeping track of the order of the appearance of start-tags in D. Since the same start-tag may appear multiple times in D, their relative positions in D are maintained in S for future reference. Each entry of TBL, which is labeled by a start-tag name s, is associated with a linked list of nodes that keeps track of the

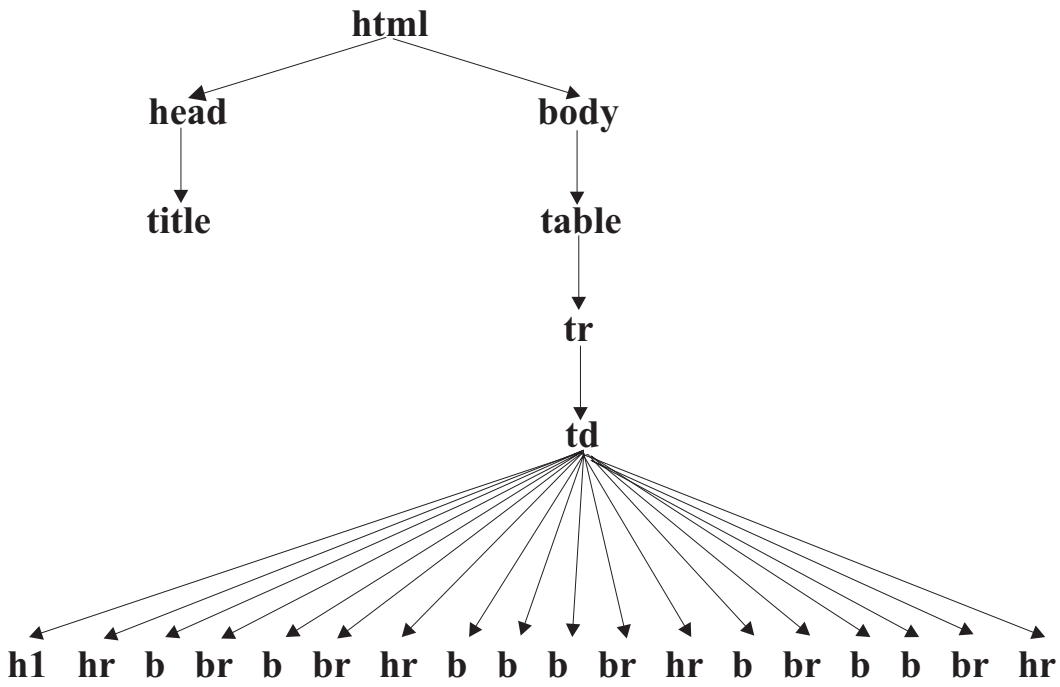

Figure 2.3: The tag tree of the Web document in Figure 2.1

positions of each appearance of s in D and S. (2) The second step scans through D to discard useless tags and insert all "missing" end-tags. In order to build the tag tree of D using a linear algorithm, we need to make sure that for every start-tag in D, its corresponding end-tag also exists in D. If the corresponding end-tag of a start-tag is "missing" in the original D, we insert the "missing" end-tag into D. S and TBL are set up for inserting missing end-tags into D and eliminating useless tags from D. All start-tags that are encountered in this pass through D are pushed onto S. Also, the relative position of each start-tag $<S_t>$ in D and its location on S are stored in a new node which is then inserted at the beginning of a linked list whose tail is pointed by the entry indexed by $<S_t>$ in TBL. (3) The final step scans through D again, which now has every "missing" end-tag. In this pass, T is created according to an in-order traversal.

Algorithm $Construct\_Tree$ has time complexity $\mathcal{O}(n)$, where n is the length of the input Web document D. Scanning D to insert all the start-tags in D as labels of

entries in table TBL takes $\mathcal{O}(n)$ time. For an appropriate list representation, adding a new entry to TBL for each new start-tag and creating a node in a (an existing) linked list takes a constant amount of time. Since we do not consider a tag more than once after it has been put in the table, discarding useless tags and inserting a "missing" end-tag into D for its corresponding start-tag which appears on the stack takes at most $\mathcal{O}(t)$ time, where t is the number of tags in D. Hence, the insertion and deletion operations are at most $\mathcal{O}(t)$. In the construction step (i.e., Step 3) of building the tag tree T of D, the number of nodes to be constructed in T is proportional to t, and the plain text to be inserted into each node is proportional to n, the length of D. Since $n > t$, algorithm $Construct\_Tree$ has time complexity $\mathcal{O}(n)$.

For example, consider the Web document D in Figure 2.1. Algorithm $Construct\_Tree$ first inserts all "missing" end-tags in D. (Note that there is no useless tag in D.) Figure 2.4 shows the document with all the "missing" end-tags inserted. (By inserting "missing" end-tags, note that we are not preparing the document for display; instead we are preparing it to help build tag trees. The updated document is discarded once the tag tree is built.) The entire tag tree of the document is shown in Figure 2.3 after the construction step. (Note that only the name of the start-tag element of each node is shown in the figure.)

2.3 The Largest Fan-out Heuristic Approach

A Web document D, which contains the records of interest, usually contains other irrelevant information such as a header and a trailer. To detect the boundary of records of interest in D, we first attempt to detect the region in the tag tree T of D that contains the records by searching for the largest-fan-out node N of T, where the records of interest in D is located in the sub-tree rooted at N (i.e., the largest-fan-out sub-tree).

Algorithm *Construct_Tree* /* Construct the tag tree of a Web document */
Input: A Web document D
Output: The tag tree T of D
Begin /* Algorithm */

/* Initialization */
1. Initialize stack S = [] and TBL := [], an array such that an entry of TBL is to be labeled by a start-tag and associated with a linked list of nodes, and let $n := 0$, where n denotes the number of entries in TBL
2. REPEAT
2.1. Locate the next tag G in D and set N := the name of G
2.2. IF G starts with <! /* a comment-tag */
THEN remove G from D /* eliminates a useless tag */
2.3. ELSE IF G is a start-tag
2.3.1. THEN IF $N \neq TABLE[j], \forall j, 0 \leq j < n$ /* G is not in TBL */
THEN $TABLE[n] := N$, $n := n + 1$, and $PUSH(N)$ /* create an entry in TBL with label N and push N onto stack S */
END-IF
2.3.2. Create a node of the form [L, S_p], where L is the location of the next tag in D and S_p is the location of N on S, and link it to the entry with label N in TBL
END-IF
2.4. ELSE /* G is an end-tag */
IF (the corresponding start-tag G_s of G does not exist in TBL)
OR (the linked list associated with G_s is *null*)
/* an end-tag without its corresponding start-tag */
THEN remove G from D /* eliminates a useless tag */
ELSE DO /* Search for the corresponding start-tag of G in S */
2.4.1. Let $A := POP()$
2.4.2. Remove the corresponding node in the linked list associated with A
/* A is no longer on the stack */
2.4.3. IF A is not the corresponding start-tag of G
THEN insert the corresponding end-tag of A at L in D, where L is the first field in the node linked to the entry A in TBL which points to A on S
END-IF
WHILE A is not the corresponding start-tag of G
END-IF
END-IF
UNTIL end-of-file(D)
3. /* Construct the tag-tree T */
/* Construct the root node R of T, which is of the form [S, I, P], where S, I, and P are the contents of the start-tag, inside-text, and post-text elements of R, respectively. */

3.1. Initialize the root node R of T such that $R := [nil, nil, nil]$ and set $cnt := 0$ and $position := 0$, where cnt is the number of sub-trees of R and $position$ is the offset from the beginning of D

3.2. WHILE $position \neq$ the location of end-of-file(D)
let (CT[cnt], $position$) := Create_Subtree(D, $position$), where CT[cnt] is the (cnt + 1)-th sub-tree of R, and $position$ is the offset from the beginning of D, and set $cnt := cnt + 1$ /* Construct the child nodes of R */
END-WHILE

4. Return T

End /* Algorithm */

Algorithm *Create_Subtree* /* Create a sub-tree of the tag tree of D */
Input: a Web document D, and POS, the offset from the beginning of D
Output: a (sub-tree of the) tag tree S, which represents (part of) D, and new offset from the beginning of D
Begin /* Algorithm */

/* Construct the root node R of T, which is of the form [S, I, P], where S, I, and P are the contents of the start-tag, inside-text, and post-text elements of R, respectively. */

1. Initialize the root node R of T such that $R := [nil, nil, nil]$ and set $cnt := 0$, where cnt is the number of sub-trees of R
2. Search for the first tag F starting from the POS-th position in D /* the tag is always a start-tag and begins at the POS-th position in D */
3. Set $R.S := F$ and $POS := POS$ + size_of($R.S$)
4. REPEAT
4.1. Search for the next tag G after F in D
4.2. Set $R.I :=$ the plain text between F and G and $POS := POS$ + size_of($R.I$)
4.3. IF G is not the end-tag of F
THEN let (CT[cnt], POS) := $Create\_Subtree$(D, POS), where CT[cnt] is the (cnt + 1)-th sub-tree of R and POS is the offset from the beginning of D and set $cnt := cnt + 1$ /* construct the child nodes of R */
END-IF
UNTIL G is the corresponding end-tag of F
5. Set $R.P :=$ the plain text between G and the next tag or the end of D if G is the last tag in D and $POS := POS$ + size_of($R.P$)
6. Return (T, POS)

End /* Algorithm */

```
1: <html><head><title>Classifieds</title></head>
2: <body bgcolor="#FFFFFF">
3: <table><tr><td>
4: <h1 align="left">Funeral Notices - </h1> October 1, 1998
5: <hr>
6: </hr><b>Lemar K. Adamson</b><br> age 84, of Tucson, died September 30, 1998.
7: He is survived by wife, Cindy; daughters, Elvia, Gloria, Irene, Isabel,
8: Jewel, and Jessica; sons, Paul, John, Jeffery, and Louis; brothers, Kirk,
9: Justin, Ivan, Hubert and Grover. Funeral service 10:00 a.m. Monday,
10: October 5, 1998 at Silverbell Ward, 1540 E. Linden. Burial in City
11: Cemetery. Friends may call from 9:00 a.m. to 10:00 a.m. Monday, at the
12: church. Arrangements by </br><b>MEMORIAL CHAPEL</b>, 236 S. Scott<br>
13: </br><hr>
14: Our beloved </hr><b>Brian Fielding Frost</b>, age 41, passed away Wednesday
15: morning, September 30, 1998, due to injuries sustained in an automobile
16: accident. He was born January 12, 1957 in Salt Lake City, to Donald
17: Fielding and Helen Glade Frost. He married Susan Fox on June 1, 1981.
18: He is survived by Susan; sons Alfred, Joseph; parents, and two sisters,
19: Anne (Dale) Elkins and Sally (Kent) Britton. Funeral services will be
20: held at 12 noon Tuesday, October 6, 1998 in the <b>Howard Stake Center</b>,
21: 350 South 1600 East. Friends may call 5-7pm. Monday at
22: <b>Carrillo's Tucson Mortuary</b>, 3401 S. Highland Drive. Interment at
23: Holy Hope Cemetery.<br>
24: </br><hr>
25: </hr><b>Leonard Kenneth Gunther</b><br> age 82. A resident of Tucson,
26: passed away peacefully on September 30, 1998. He was born June 6, 1916 in Iowa.
27: He joined the U.S. Navy serving during World War II. He remained a member
28: of the U.S. Naval Reserve (USNR) for several years. He is survived by his wife,
29: Gwendolyn; sons, Eric D. of San Francisco, CA, Vincent J. of Tucson; a daughter,
30: Janet H. of Provo, UT; and one granddaughter, Sarah R. of Phoenix, AZ.
31: Friends may call from 5:00 p.m. until 7:00 p.m. on Monday, October 5, 1998 at
32: </br><b>HEATHER MORTUARY</b>, 1040 N. Columbus Blvd. Funeral services
33: will be at 11:00 a.m. at <b>HEATHER MORTUARY</b>, on Tuesday,
34: October 6, 1998. Burial will be private at South Lawn Cemetery.<br>
35: </br><hr>
36: </hr></td></tr></table>
37: All material is copyrighted.
38: </body>
39:</html>
```

Figure 2.4: A sample Web document after all the "missing" end-tags have been inserted

Algorithm *Find_Largest_Fanout* /* Finds the largest-fan-out node of a tag tree */
Input: (a sub-tree of) the tag tree T of a Web document D
Output: the largest-fan-out node S in T and max, the fan-out of S
Begin /* Algorithm */

1. Set $S :=$ the root node R of T and $max :=$ the fan-out of R
2. IF $max > 0$ /* R has at least one child node */
   THEN
      FOR each sub-tree u rooted at a child node of R DO
2.1. Set $(S', max') := Find\_Largest\_Fanout(u)$
2.2. IF $max' > max$
      THEN set $S := S'$ and $max := max'$
      END-IF
      END-FOR
   END-IF
3. Return S and max

End /* Algorithm */

Algorithm *Find_Largest_Fanout* details the step-by-step process of locating the largest-fan-out node of T. This algorithm, which takes T as an input and returns the largest-fan-out node S of T and max, which is the fan-out of S, consists of two main steps. In the first step, S is initialized to be the root node R of T, and the largest number of fan-out, max, is initialized to be the fan-out of R. In the second step, algorithm *Find_Largest_Fanout* is recursively called for each node of T, except R, to locate the largest-fan-out node in T, R included.

The complexity of algorithm *Find_Largest_Fanout* is $\mathcal{O}(t)$, where t is the number of start-tags in D. If there are t start-tags in D, then there are t nodes in T since each node contains one and only one start-tag in its start-tag element. Since algorithm *Find_Largest_Fanout* is (recursively) called for (i.e., visits) each node of T once in order to find the largest-fan-out node, the complexity of the algorithm is $\mathcal{O}(t)$.

For example, consider the Web document D in Figure 2.1 and its tag tree T in Figure 2.3. Algorithm *Find_Largest_Fanout* first initializes S to be the root node

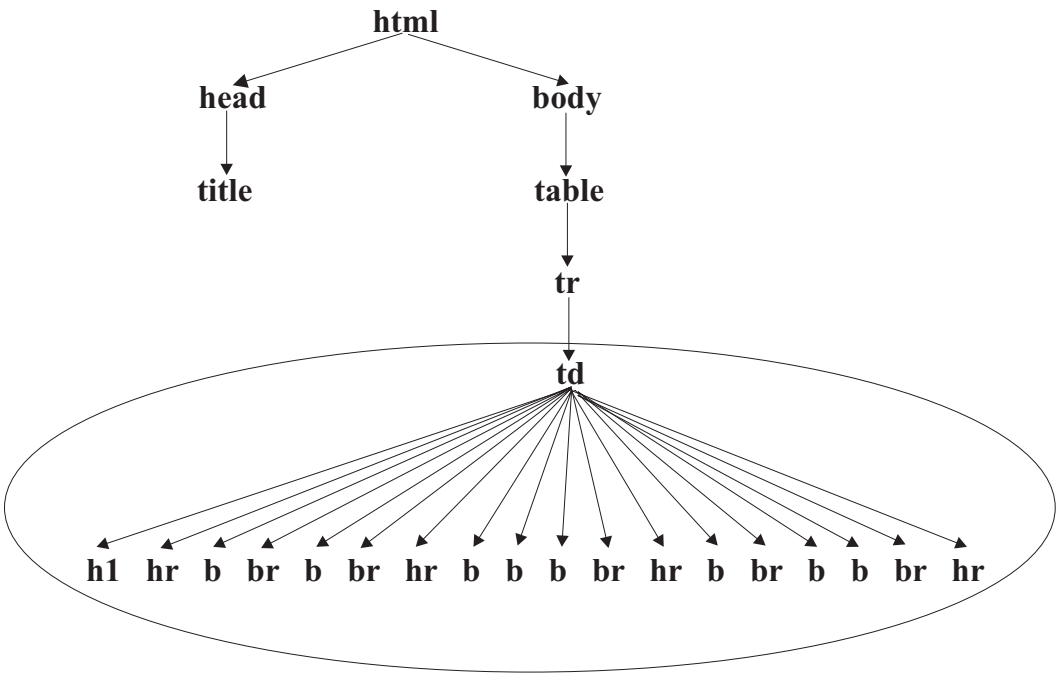

Figure 2.5: The tag tree of the Web document in Figure 2.1 with the largest-fan-out sub-tree rooted at node *td* which is embedded within an ellipse

html in T and max to be 2, the fan-out of node *html*. For node *head*, which is a child node of node *html*, and node *title*, which is the only child node of node *head*, neither one of them has larger fan-out than node *html*. Thus, S and max remain unchanged. For node *body* and its descendant nodes, $Find\_Largest\_Fanout$ detects that the fan-out of node *td* (at level 4 of T) is 18, which is greater than max. Eventually, $Find\_Largest\_Fanout$ returns *td*, which has the largest fan-out in T, and 18, the fan-out of *td*. Figure 2.5 shows the sub-tree rooted at *td* embedded within an ellipse.

2.4 The Pre-processing of the Tags

Using the largest-fan-out sub-tree rooted at node S, the number of appearance of each distinct tag name in the start-tag elements of all the child nodes of S can be determined. We distinguish all irrelevant tags from all the candidate tags using a

threshold t, which is set to be 10% of the fan-out of S. We believe that a start-tag is highly unlikely a record separator if it appears less than 10% among all the child nodes of S. If the number of appearance of a particular start-tag g is less than or equal to t, then g is treated as an irrelevant tag. All start-tags that are not irrelevant are candidate tags, because these tags become our candidates as the record separator. Algorithm *Pre-Process_Tags* determines all the candidate tags and eliminates all irrelevant tags in the child nodes of S.

In order to count the number of distinct start-tag names appeared in the child nodes of S, algorithm *Pre-Process_Tags* scans through all the child nodes of S once. Also, *Pre-Process_Tags* compares the number of appearance of each distinct start-tag name with the threshold t and eliminates all the irrelevant tags. Thus, the complexity of *Pre-Process_Tags* is $\mathcal{O}(m + s)$, where m is the fan-out of S and s is the number of distinct start-tags in the child nodes of S.

For example, consider the sub-tree rooted at the largest-fan-out node *td* in Figure 2.5. Algorithm *Pre-Process_Tags* constructs TL = [(h1, 1), (hr, 4), (b, 8), (br, 5)]. (The second element of each ordered pair in TL denotes the number of appearance of the first element among the child nodes of *td*.) Since the default t is 10% $\times$ 18 $\cong$ 2, 'h1' (which appears only once) is an irrelevant tag, and (h1, 1) is removed from TL. The other three tags, 'hr', 'b', and 'br', are candidate tags and will be further considered as the record separator in the algorithms presented in Chapters 3 and 4.

Algorithm *Pre-Process_Tags* /* Determines all the candidate tags and eliminates all irrelevant tags in the input largest-fan-out sub-tree */
Input: the sub-tree rooted at the largest-fan-out node S
Output: TL, an array of ordered pairs of the form (STR, NUM) /* STR is the name of a candidate tag and NUM is the number of appearance of its corresponding STR in the child nodes of S */
Begin /* Algorithm */

/* Initialization */
1. Initialize TL := [], t := 10% * the fan-out of S, and s := 0
/* t is the threshold to determine irrelevant tags, whereas s denotes the number of distinct start-tags in the child nodes of S */
/* Construct a tag list for all the distinct tag names */
2. FOR each child node C of S DO
2.1. Let E be the name in the start-tag element of C
2.2. IF $E \neq TL[j].STR$, $\forall j$, $0 \leq j < s$
THEN set $TL[s].STR := E$, $TL[s].NUM := 1$, and $s := s + 1$
ELSE set $TL[p].NUM := TL[p].NUM + 1$, where $TL[p].STR = E$
/* p is the location in TL where E occurs */
END-IF
END-FOR
/* Eliminate all irrelevant tags */
3. FOR k := 0 TO s - 1 DO
IF $TL[k].NUM \leq t$ /* $TL[k].STR$ is an irrelevant tag */
THEN remove $TL[k]$ from TL and set $s := s$ - 1
/* remove all irrelevant tags from TL */
END-IF
END-FOR
4. Return TL

End /* Algorithm */

Chapter 3

The Individual Heuristics

To discover the record separator of a Web document D we first apply each of the five independent heuristics (to be introduced in Sections 3.1 - 3.5) to determine the ranking of each candidate tag and then apply a combined heuristic (which will be introduced in Chapter 4) to determine the record separator. The *ranking* of a candidate tag, determined by an individual heuristic HP, is a prioritized choice of HP. For example, if HP ranks a candidate tag $<A>$ to be 1, then $<A>$ is considered by HP to be the first choice as the record separator of D. Note that several candidate tags may have the same ranking. Hence, the output of each individual heuristic is a candidate list CL, which is an array of ordered pairs in the form of (STR, NUM), where STR is the name of a candidate tag and NUM is the ranking of STR determined by the heuristic.

The five individual heuristics, ontology-matching (OM), repeating-tag pattern (RP), standard deviation (SD), identifiable "separator" tags (IT), and highest-count tags (HT), span a broad range of possible techniques for discovering record boundaries in a Web document. The OM heuristic considers the content of a record. Items that are in a one-to-one correspondence or are functional with respect to the entity of interest tend to appear once and only once in a record. If we can recognize these items, we can look for candidate tags that best separate these items into individ-

ual records. The RP heuristic makes use of the observation that divisions between records often include several tags that consistently appear in the same order. The SD heuristic is based on the observation that when multiple records about an entity appear in a document, the records are typically about the same size. Thus, the candidate tag c with the minimum standard deviation, based on the size of all the plain text appeared between each pair of c, tends to be the record separator. The IT heuristic uses a predetermined list of likely HTML separator tags. This heuristic is applicable since both hand-created HTML documents and tool-generated HTML documents tend to consistently use some common separator tags (e.g., 'hr'). The HT heuristic simply ranks the candidate tags based on the number of their appearances in the child nodes of the largest-fan-out node. This heuristic is based on the assumption that the record separator probably appears the most in a Web document that contains a large number of records.

3.1 The Ontology-Matching Heuristic

An application *ontology* describes the application of interest. It is a conceptual model augmented with additional information to describe constants and keywords for the application. We develop the ontological model instance for the domain of interest. In this thesis, our ontologies are assumed to be *narrow in breadth* (meaning that the ontology is small, having no more than a few dozen object and relationship sets in its conceptual model) and that our target Web documents are assumed to be *rich in data* (meaning that there is an abundance of recognizable constants such as email addresses, phone numbers, names of automobile makes and models, and so forth). Figure 3.1 shows the ontological model instance for the obituaries application in graphical form, and the entity of interest is *Deceased Person* that is marked by "-> •" as shown in the figure. We adopt the Object-oriented Systems Model (OSM)

[EKW92] to describe our ontology.

In OSM rectangles represent sets of objects. Dotted rectangles and solid rectangles represent lexical object sets and non-lexical object sets, respectively. Whether an object set is lexical or non-lexical depends on whether its associated data frame ([Emb80]) describes a set of possible strings as objects for the object set. If the data frame is for a lexical object set, it describes the string patterns for its constants. Whether lexical or non-lexical object sets, an associated data frame describes context keywords that indicate the presence of an object in an object set. For example, "died" or "passed away" can be included as context keywords for *Death Date*. In OSM lines connecting rectangles represent sets of relationships, and participation constraints near connection points between object sets and relationship sets designate the minimum and maximum number of times an object in the set participates in the relationship. Binary relationship sets have a verb phrase and reading-direction arrow, and n-ary relationships have a diamond and a full descriptive name that includes the names of its connected object sets. For example, "*Funeral is on Funeral Date*" names the relationship set between *Funeral* and *Funeral Date*. In OSM a colon after an object-set name denotes that the object set is a specialization. For example, "*Birth Date: Date*" denotes that the set of objects in *Birth Date* is a subset of the objects in *Date* object set.

For our ontologies, an ontological model instance provides both a global view (e.g., across all obituaries) and a local view (e.g., for a single obituary). We express the global view as previously explained and specialize it for a particular instance by imposing additional constraints, which denotes by the "becomes" arrow (->). In Figure 3.1, for example, the *Deceased Person* object set becomes a single set, as denoted by "-> •", and the *1..** participation constraint on both *Deceased Name* and *Relative Name* becomes *1*. Thus, we declare in our ontology that an obituary

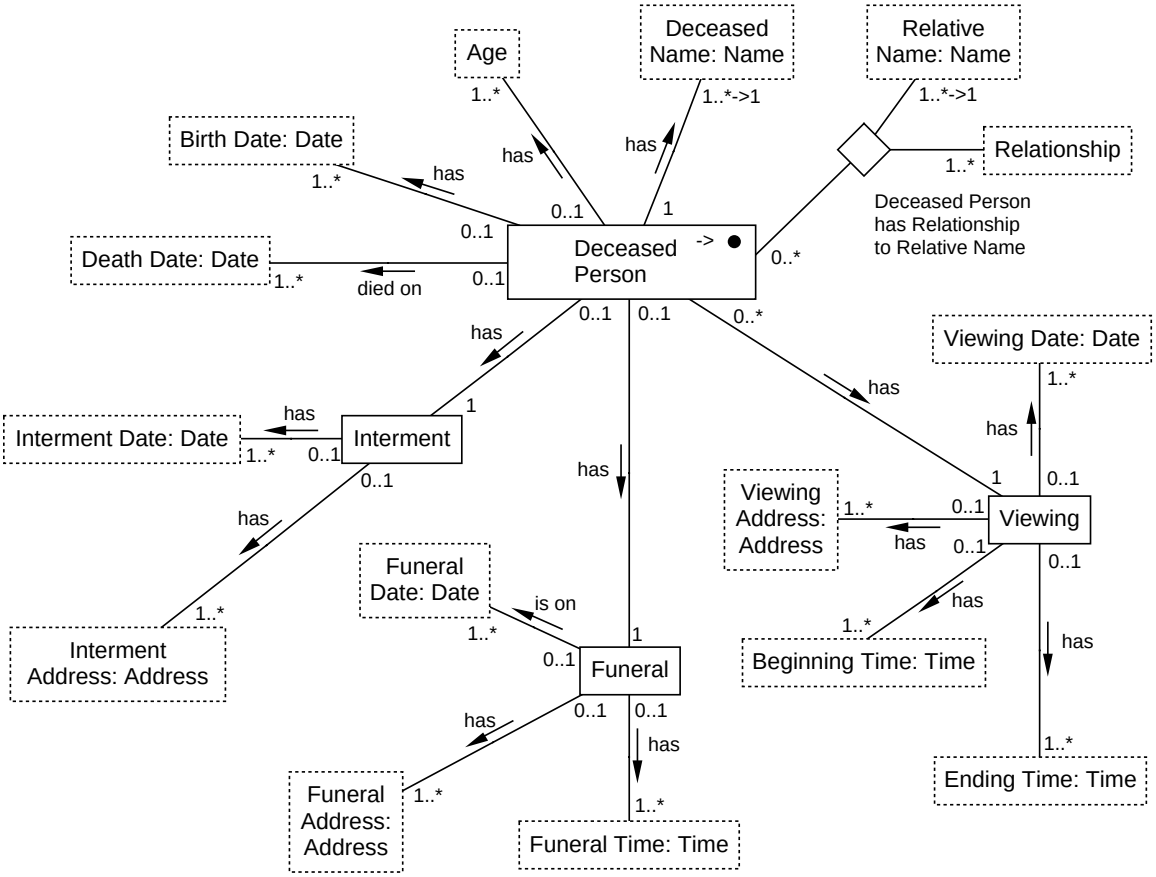

Figure 3.1: A sample ontology in graphical form

is for one deceased person and that a name either identifies the deceased person or the family relationship of a relative of the deceased person. From these specializing constraints, we can also derive other facts about individual obituaries, such as there is only one funeral and one interment, although there may be several viewings and several relatives.

We may expect one or more fields, called *record-identifying field*s, of a record appear once and only once in the record. For each record-identifying field, if we can locate a value for the field or even just an indication that the value exists, we can count the number of such occurrences. Then, if we take the average number of occurrences for several record-identifying fields in a Web document D, we have a good chance of correctly estimating the number of records in D. With this estimation, we

rank the candidate tags by how closely their number of appearances corresponds to the estimated number of records. For example, the *Death Date* for an obituary is a record-identifying field because there should be one and only one death date in each record. As an indication that this field exists, we use a keyword set that includes "died on" and "passed away on" to indicate the existence of the field. We do not use the date itself because there may be many other fields in the record such as *Birth Date* and *Funeral Date* that are also dates. Although date values themselves are not record-identifying indicators for obituaries, the keywords that distinguish among the various dates are excellent indicators for the existence of record-identifying fields. We note that a record-identifying field is not the same as a key for a record, but rather is a field that is likely to occur once and only once for each record. A death date, for example, occurs once in every obituary, but a death date is not a key that identifies deceased persons in a genealogical database.

A given application ontology contains the information needed to determine the record-identifying fields. All object sets whose objects have a one-to-one correspondence with the entity of interest designate record-identifying fields as well as all object sets whose objects are functionally dependent on the entity of interest. We are selective in choosing which record-identifying fields to consider in our ontology-matching (OM) heuristic. We limit the number of fields to be at least 3 and no more than 20% of the number of sets of objects in the ontology. We want at least 3 so that we can obtain a reasonable average for estimating the number of records in a Web document. (If we do not have at least 3 record-identifying fields, we do not use our OM heuristic.) We also set an upper bound because we want to use only a few of the "best" record-identifying fields. For example, consider the obituaries application ontology shown in Figure 3.1, there are 19 sets of objects (i.e., the number of rectangles). Thus, the number of record-identifying fields that we are interested in should be 4 (i.e., $max(3,$

```
KEYWORD(Interment) : \bburial\b
KEYWORD(Interment) : \binterment\b
KEYWORD(Funeral)   : \brosary\b
KEYWORD(Funeral)   : \blaid\s+to\s+rest\b
KEYWORD(Funeral)   : \bmass(\s+of\s+christian\s+burial)\b
KEYWORD(Funeral)   : \bcelebration\s+of\s+(his|her)\s+life\b
KEYWORD(Funeral)   : \bprivate\s+service
KEYWORD(Funeral)   : \bmemorial\s+service
KEYWORD(Funeral)   : \bgraveside\s+service
KEYWORD(Funeral)   : \bfuneral(?!\s+(home|director))\b
KEYWORD(BirthDate) : \bborn\b
KEYWORD(DeathDate) : \breturned\s+to\b
KEYWORD(DeathDate) : \bjoined\s+(his|her)\b
KEYWORD(DeathDate) : \bhave\s+gone\s+together\b
KEYWORD(DeathDate) : \bfinished\s+life
KEYWORD(DeathDate) : \bsuffering\s+is\s+over\b
KEYWORD(DeathDate) : \bleft\s+(his|her)\s+mortal
KEYWORD(DeathDate) : \bwas\s+called\s+home\b
KEYWORD(DeathDate) : \bwas\s+released\s+from\b
KEYWORD(DeathDate) : \bpassed\s+over\b
KEYWORD(DeathDate) : \bpassed\s+on\b
KEYWORD(DeathDate) : \bpassed\s+away\b
KEYWORD(DeathDate) : \bdied\b
```

Figure 3.2: The ontology rules for the obituaries application

$19 \times 20\%) = max\ (3,4) = 4$). We order the potential record-identifying fields from "best" to "worst" by first considering fields that are in a one-to-one correspondence with the entity of interest and then considering those that are functionally dependent on the entity of interest. Then, within these fields we consider keyword indicators first followed by identifiable values, except that we do not consider identifiable values that share a common type (e.g., dates in the obituary example). For example, we have chosen the keyword indicators associated with the sets of objects *Internment*, *Funeral*, *Birth Date*, and *Death Date* in Figure 3.1 as our input ontology rules since these sets of objects have one-to-one relationships with the entity of interest *Deceased Person*. The ontology rules are for keywords associated with these object sets represented in regular expressions as shown in Figure 3.2.

To apply our OM heuristic, we first count the number of appearance of each distinct record-identifying field in a Web document D and calculate the average number of appearance of all record-identifying fields in D. We then consider the number of appearance of each candidate tag and rank them in order by how close they come to the average. If among all the candidate tags the number of appearance of a candidate tag DT is the closest to the average number of appearances of all the record-identifying fields in D, then the ranking of DT is 1 (i.e., the first choice to be the record separator).

Algorithm *Apply_OM* describes the step-by-step process of implementing the OM heuristic. The *Constant/Keyword Recognizer* ([ECJ+98]) applies the ontology rules to the plain text extracted from the largest-fan-out sub-tree and produces a data-record table DRT. Each row e in DRT, which describes either a constant or a keyword, consists of four fields separated by bars ("|"). The first field of e is a descriptor such that if e describes a constant, then the descriptor is an object-set name to which the constant may belong, and if e describes a keyword, then the descriptor is of the format $KEYWORD(w)$, where w is an object-set name to which the keyword may apply. The second field of e is the constant or keyword found in the document. The last two fields give the positions of the beginning and ending character locations in the document for the first and last symbol of the recognized constant or keyword (i.e., the second field), respectively. In our OM heuristic, we only need the first two fields. *Apply_OM* categorizes the content of DRT based on the descriptors, calculates the average number of appearances of all the distinct descriptors, and computes the absolute value of the difference between the number of appearance of each candidate tag and the average number of appearance of all the distinct descriptors. Finally, *Apply_OM* generates the candidate list CL, which orders all the candidate tags according to their rankings such that the candidate tag

with the smallest absolute value is ranked as 1 and placed at the beginning of CL.

We check for the existence of a keyword or constant value by matching a regular expression with the plain text in the largest-fan-out sub-tree. Since this matching process is at best $\mathcal{O}(pr)$, where p is the size of the plain text and r is the number of regular expressions, the running time of the OM heuristic is not linear. We observe, however, that in the overall data-extraction process, as shown in Figure 1.1, we must run the regular expressions over all the plain text in the largest-fan-out sub-tree. We further observe that if we integrate processes, we can run the regular-expression matching process before separating records at no additional cost. This is because the entries in the *Data-Record Table* are ordered by position in the document. (Figure 3.3 shows a portion of the *Data-Record Table* for the Web document in Figure 2.1 in the overall process.) Once we discover the record separator, we can use the position of the record separator in the document to partition the *Data-Record Table* into sets of entries that are in a one-to-one correspondence with the records, and use these sets of entries for further downstream processing by the *Database-Instance Generator* (see Figure 1.1). Using this approach, we assume the regular-expression matching for the OM heuristic has already been done for the overall data-extraction process and thus ignore its complexity measure in the OM heuristic. Since the *Data-Record Table* contains all the recognized keywords and values, along with their associated object sets and their positions within the plain text of the document, a single scan through the table allows us to obtain the counts we need. Thus, the OM heuristic is $\mathcal{O}(d)$, where d is the number of lines in the *Data-Record Table* for the plain text in the largest-fan-out sub-tree. Although d may be large, for practical cases it is not typically larger than n, the document size; thus, we assume $\mathcal{O}(d) < \mathcal{O}(n)$. Also, we note that sorting the candidate tags takes $\mathcal{O}(s\ log\ s)$ time, where s is the number of candidate tags. Since $s \ll d$, the sorting operation is negligible.

KEYWORD(Funeral)|Funeral|1|7
KEYWORD(DeathDate)|died|73|76
KEYWORD(Funeral)|Funeral|281|287
KEYWORD(Interment)|Burial|372|377
KEYWORD(DeathDate)|passed away|565|575
KEYWORD(BirthDate)|born|677|680
KEYWORD(Funeral)|Funeral|919|925
KEYWORD(Interment)|Interment|1125|1133
KEYWORD(DeathDate)|passed away|1215|1225
KEYWORD(BirthDate)|born|1268|1271
KEYWORD(Funeral)|Funeral|1731|1737
KEYWORD(Interment)|Burial|1822|1827

Figure 3.3: The Data-Record Table generated by using the ontology rules in Figure 3.2

For example, consider the Web document in Figure 2.1, the sub-tree rooted at the largest-fan-out node *td* as shown in Figure 2.5, and TL = [(hr, 4), (b, 8), (br, 5)] as computed in Section 2.4. In TL, there are 4 hr's, 8 b's, and 5 br's which appear in the child nodes of node *td*. Using the ontology rules in Figure 3.2, the *Constant/Keyword Recognizer* generates the *Data-Record Table* DRT as shown in Figure 3.3. Algorithm *Apply_OM* then counts the number of appearance of each distinct descriptor, and the result is shown as follows, where the second element is the number of appearance of its corresponding first element:

$(KEYWORD(Funeral), 4)$,

$(KEYWORD(DeathDate), 3)$,

$(KEYWORD(Interment), 3)$,

$(KEYWORD(BirthDate), 2)$.

Thus, the average number of appearance of all the descriptors is three (i.e., (3 + 3 + 3 + 2) / 4 = 3). After calculating the absolute values of the differences between the appearance of each candidate tag in TL and three, TL becomes [(hr, 1), (b, 5), (br, 2)], where the absolute value appears as the second element in each ordered pair of TL. Since $1 < 2 < 5$, the computed CL = [(hr, 1), (br, 2), (b, 3)].

Algorithm *Apply_OM*

Input: the largest-fan-out sub-tree S, the ontology rules $CKMR$, and TL, an array of ordered pairs in the form of (STR, NUM), which contains all the distinct candidate tags and their numbers of appearances in the child nodes of the root node in S. /* TL is generated by algorithm *Pre-Process_Tags* */

Output: the candidate list CL that contains all the candidate tags and their rankings

Begin /* Algorithm */

1. Let PT := *Get_Plaintext*(S) /* extracts plain text from the largest-fan -out sub-tree */
2. Call *Constant/Keyword Recognizer*(PT, $CKMR$) which returns the *Data-Record Table* DRT, where each row of DRT contains a *descriptor*, a *string*, and the starting and ending *positions*
   /* Determines the content of each *descriptor* element in DRT */
3. Initialize $n := 0$ and $DT = []$ /* DT is an array of ordered pairs in the form of (STR, NUM) */
4. FOR each entry e of DRT DO
   IF $e.descriptor \neq DT[j].STR$, $\forall j$, $0 \leq j < n$
   THEN set $DT[n].STR := e.descriptor$, $DT[n].NUM := 1$,
   and $n := n + 1$
   ELSE set $DT[p].NUM := DT[p].NUM + 1$, where $DT[p].STR =$ $e.descriptor$ /* p is the location in DT where $e.descriptor$ occurs */
   END-IF
   END-FOR
   /* Calculate the average number of appearance of all the keys */
5. Initialize $sum := 0$
6. FOR $j := 0$ TO n - 1 DO
   $sum := sum + DT[j].NUM$
   END-FOR
7. Set $ave := sum$ / n
   /* Calculate the difference between the number of appearance of a tag and ave */
8. FOR $j := 0$ TO $|TL|$ - 1 DO
   $TL[j].NUM := abs(TL[j].NUM$ - $ave)$ /* abs is the absolute value of a number */
   END-FOR
9. TL := sort(TL) so that for each j and k such that $0 \leq j < k < |TL|$, $TL[j].NUM \leq TL[k].NUM$ /* the tag with the smallest difference is placed at the beginning of the list */

/* Construct the candidate list such that the tag with the smallest difference has the highest ranking, i.e., 1 */

10. Initialize $r := 0$ /* the counter of the rankings */
11. FOR $m := 0$ TO $|TL|$ - 1 DO /* determines the ranking of each candidate tag */
    11.1. Set $CL[m].STR := TL[m].STR$ and $r := r + 1$
    11.2. IF $m > 0$ AND $TL[m].NUM = TL[m-1].NUM$
    THEN $CL[m].NUM := CL[m-1].NUM$ /* assigns the same ranking to the two candidate tags which have the same absolute value */
    ELSE $CL[m].NUM := r$ /* assigns ranking r to the candidate tag $CL[m].STR$ */
    END-IF
    END-FOR
12. Return CL

End /* Algorithm */

Algorithm *Get_Plaintext* /* extracts the plain text from the given tag tree */

Input: a (sub-tree of the) tag tree T rooted at node R

Output: the plain text PT that is in the region bounded by T

Begin /* Algorithm */

1. Set $PT :=$ []
2. Set $PT :=$ append ($R.I$, PT) /* $R.I$ is the content of the *inside-text* element of R */
3. FOR each sub-tree rooted at a child node c of R, chosen according to the left-to-right order, DO
    3.1. Set $PT' :=$ *Get_Plaintext*(c)
    3.2. Set $PT :=$ append(PT', PT) /* appends PT' to PT */
    END-FOR
4. Set $PT :=$ append ($R.P$, PT) /* $R.P$ is the content of the *post-text* element of R */
5. Return PT

End /* Algorithm */

3.2 The Repeating-Tag Pattern Heuristic

Our repeating-tag pattern (RP) heuristic is based on the observation that there often exist consistent patterns of two or more adjacent tags at the record boundaries in a Web document. For example, some candidate tags may consistently appear before or after the record separators. If a pair of candidate tags <a> <b> (resp. <b> <a>) occurs at a record boundary and <a> is the record separator, the number of the appearance (i.e., count) for this pair should be the same as the count of the number of occurrence of <a> alone. For simplicity, we only consider the number of appearance of each pair of distinct adjacent candidate tags.

Our RP heuristic first counts the number of occurrence of each pair of candidate tags that have no intervening plain text. For each of these pairs <a> <b>, the RP heuristic calculates the absolute value of the difference between the count of the pair and the count of <a> alone (resp. <b> alone). Finally, the candidate tags are ranked in ascending order according to their absolute values (i.e., the candidate tag with the smallest absolute value among all the candidate tags is ranked 1). Since a particular candidate tag may appear more than once in difference pairs (i.e., may have several absolute values), our approach discards all but the smallest of the absolute values of the candidate tag. Note that our RP heuristic may not supply the rankings of the candidate tags if there does not exist any pair of adjacent candidate tags that have no intervening plain text in the child nodes of the largest-fan-out node. Algorithm *Apply_RP* details our RP heuristic.

To analyze the running time of algorithm *Apply_RP*, we first observe that we can make a single pass through the plain text in the descendent nodes of the largest-fan-out sub-tree S and create up to s^2 pairs of adjacent candidate tags in $\mathcal{O}(e + s^2)$ time, where e is the number of plain text characters in S and s is the number of distinct

candidate tags. Taking the absolute value of the difference between the count of a pair and the count of each candidate tag in the pair and checking the candidate tag as a record separator requires a pass through all the pairs, an $\mathcal{O}(s^2)$ operation. Sorting each of the candidate tags in ascending order on their absolute values and removing duplicates take as much as $\mathcal{O}(s^2\ log\ s^2)$ time. Thus, the complexity of algorithm *Apply_RP* is $\mathcal{O}(s^2(2 + log\ s^2) + e)$. Since $e \gg s$, we can neglect all the operations on s and obtain $\mathcal{O}(e)$ as the time complexity of algorithm *Apply_RP*. Since $e \leq n$, where n is the size of the Web document, algorithm *Apply_RP* is bounded by $\mathcal{O}(n)$.

For example, consider the largest-fan-out sub-tree rooted at node *td* in Figure 2.5 and TL = [(hr, 4), (b, 8), (br, 5)] as computed in Section 2.4. Since the numbers of occurrences of the pairs "hr b" and "br hr" are 2 and 3, respectively, $PAIR\_TBL$ = [("hr b", 2), ("br hr", 3)] is created by algorithm *Apply_RP*. The absolute value of the difference between the count for "hr b", i.e., 2, and the count for 'hr', i.e., 4, is 2. The absolute value of the difference between the count for "br hr", i.e., 3, and the count for 'hr', i.e., 4, is 1. Since the algorithm takes the smallest number if a candidate tag appears more than once in difference pairs in $PAIR\_TBL$, the absolute value of the difference is 1 for 'hr'. After calculating the absolute values for 'b' and 'br', *Apply_RP* generates L = [(hr, 1), (b, 6), (br, 2)]. Since $1 < 2 < 6$, CL = [(hr, 1), (br, 2), (b, 3)], as computed by algorithm *Apply_RP*.

3.3 The Standard Deviation Heuristic

Standard deviation measures the variability of a set of numbers deviated from their average and reflects the contribution of all numbers. The value of the standard deviation of the numbers is independent of the size of the set of numbers. In the standard deviation (SD) heuristic we determine the record separator based on the

Algorithm *Apply_RP*

Input: the largest-fan-out sub-tree S and TL, an array of ordered pairs of the form (STR, NUM), where STR is the name of a candidate tag and NUM is the number of appearance of STR in the child nodes of the largest-fan-out node /* S and TL are generated by algorithms $Find\_Largest\_Fanout$ and $Pre\text{-}Process\_Tags$, respectively */

Output: the candidate list CL that contains all the candidate tags and their rankings

Begin /* Algorithm */

1. Initialize $PAIR\_TBL$:= [] /* $PAIR\_TBL$ is an array of ordered pairs of the form (STR, NUM), where STR is a pair of adjacent candidate tags and NUM is the number of the appearance of STR in the child nodes of the root node in S */
2. FOR j := 1 TO the fan-out of the root node R in S DO
2.1. Set tag_1 := the name of the start-tag in the j-th child node of R and tag_2 := the name of the start-tag in the (j+1)-th child node of R
2.2. IF $tag_1 = TL[x].STR$ AND $tag_2 = TL[y].STR$, where $0 \leq x, y < |TL|$ /* both tag_1 and tag_2 are candidate tags */
THEN IF the fan-out of the j-th child node of R is 0 /* leaf node */
2.2.1. THEN set txt := $Get\_Plaintext$(the j-th child node of R)
2.2.2. IF $|txt| = 0$ OR $txt[z] =$ ' ', $\forall z$, $0 \leq z < |txt|$
/* tag_1 and tag_2 are adjacent */
THEN $PAIR\_TBL$:= $Add\_Pair(PAIR\_TBL, tag_1, tag_2)$
END-IF
END-IF
END-IF
END-FOR
3. Initialize $n := 0$ and L := [] /* L is an array of ordered pairs of the form (STR, NUM), where each entry stores the name of a candidate tag and its smallest absolute value */
4. FOR $j := 0$ TO $|PAIR\_TBL|$ - 1 DO
4.1. Set tg_1 := the string that appears before ' ' in $PAIR\_TBL[j].STR$ and tg_2 := the string that appears after ' ' in $PAIR\_TBL[j].STR$
/* calculates the absolute value of the difference between the count for the pair and the count for tg_1 alone */
4.2. Find position i in TL such that $TL[i].STR = tg_1$, $0 \leq i < |TL|$ and set num_1 := abs($TL[i].NUM$ - $PAIR\_TBL[j].NUM$)

/* calculates the absolute value of the difference between the count for the pair and the count for tg_2 alone */

4.3. Find position i in TL such that $TL[i].STR = tg_2$, $0 \leq i < |TL|$ and set num_2 := abs($TL[i].NUM$ - $PAIR\_TBL[j].NUM$)

4.4. IF $tg_1 \neq L[k].STR$, $\forall k$, $0 \leq k < n$
THEN set $L[n].STR := tg_1$, $L[n].NUM := num_1$, and $n := n + 1$
ELSE IF $L[p].NUM > num_1$, where $L[p].STR = tg_1$ /* p is the location in L where tg_1 occurs */
THEN $L[p].NUM := num_1$ /* replaces smaller abs. value */
END-IF
END-IF

4.5. IF $tg_2 \neq L[k].STR$, $\forall k$, $0 \leq k < n$
THEN set $L[n].STR := tg_2$, $L[n].NUM := num_2$, and $n := n + 1$
ELSE IF $L[p].NUM > num_2$, where $L[p].STR = tg_2$ /*p is the location in L where tg_2 occurs */
THEN $L[p].NUM := num_2$ /* replaces smaller abs. value */
END-IF
END-IF
END-FOR

5. Set L := sort(L) so that for each j and k such that $0 \leq j < k < |L|$, $L[j].NUM \leq L[k].NUM$ /* the tag with the smallest absolute value is placed at the beginning of L */

6. Initialize CL := [] and $r := 0$ /* r is the counter of the rankings */

7. FOR $m := 0$ TO $|CL|$ - 1 DO /* determines the ranking of each candidate tag */

7.1. Set $CL[m].STR := L[m].STR$ and $r := r + 1$

7.2. IF $m > 0$ AND $L[m].NUM = L[m-1].NUM$
THEN $CL[m].NUM := CL[m-1].NUM$ /* assigns the same ranking to the two candidate tags which have the same absolute value */
ELSE $CL[m].NUM := r$ /* assigns ranking r to the candidate tag $CL[m].STR$ */
END-IF
END-FOR

8. Return CL

End /* Algorithm */

Algorithm *Add_Pair* /* Adds a pair of tags into an entry of the input table or increase the count of the entry where the pair locates */

Input: $PAIR\_TBL$, and tag_1 and tag_2, where $PAIR\_TBL$ is an array of ordered pairs of the form (STR, SUM), where STR is a pair of candidate tags and SUM is the number of appearance of STR in the adjacent child nodes of the largest-fan-out node, and tag_1 and tag_2 are two adjacent candidate tags

Output: $PAIR\_TBL$, an updated version of the input table $PAIR\_TBL$

Begin /* Algorithm */

1. Let $n := |PAIR\_TBL|$ and $tg :=$ concat(tag_1, ' ', tag_2) /* n is the number of entries in $PAIR\_TBL$ and tg is the result of combining tag_1 and tag_2 together with a space in between */
2. IF $tg \neq PAIR\_TBL[j].STR$, $\forall j$, $0 \leq j < n$ /* the pair does not exist in the table */
   THEN set $PAIR\_TBL[n].STR := tg$, $PAIR\_TBL[n].NUM := 1$
   ELSE set $PAIR\_TBL[p].NUM := PAIR\_TBL[p].NUM + 1$, where $PAIR\_TBL[p].STR = tg$ /* p is the location in $PAIR\_TBL$ where tg occurs */
   END-IF
3. Return $PAIR\_TBL$

End /* Algorithm */

assumption that the records of interest often have approximately the same length, and thus the standard deviation of the intervals (in terms of the number of characters) between the record separators should be smaller than that of the other candidate tags.

The SD heuristic is detailed in algorithm *Apply_SD*, which takes the largest-fan-out sub-tree and a list of candidate tags (which is generated by algorithm *Pre-Process_Tags*) as input and returns the candidate list. *Apply_SD* first calls algorithm *Calculate_SD*, which calculates the standard deviation of the lengths of the plain text lying between each pair of consecutive, identical candidate tags[1]. It then determines the ranking of each candidate tag based on the standard deviation of the lengths. The candidate tag which has the smallest standard deviation is ranked as 1.

Algorithm *Calculate_SD* first calculates the length of the plain text between each pair of consecutive, identical candidate tags T and places all the lengths into an array $INTERVAL$. Hereafter, it calculates the standard deviation of the lengths, denoted $SD(T)$, using the formula $((\sum_{k=0}^{n}(X_k - ave(INTERVAL))^2)/(n+1))^{1/2}$ [Wit89], where X_k is an element of $INTERVAL$ $(0 \leq k \leq n)$, the number of elements in $INTERVAL$ is $n+1$, and ave($INTERVAL$) is the average of the elements in $INTERVAL$.

The complexity of algorithm *Apply_SD* is $\mathcal{O}(e)$, where e is the length of the plain text in the descendent nodes of the largest-fan-out sub-tree S. In order to calculate $SD(T)$ of each distinct candidate tag T, algorithm *Calculate_SD* retrieves the plain text in S, which is of length e. Thus, the complexity of *Calculate_SD* is $\mathcal{O}(e)$. *Apply_SD* calls *Calculate_SD* for each distinct candidate tag and sorts all the distinct candidate tags according to the standard deviations of the lengths for the candidate tags to obtain the rankings. Hence, the complexity of *Apply_SD* is

[1]A pair of consecutive, identical candidate tags is a pair of identical tags which appear one after the other regardless whether there exist other distinct tags or plain text in between.

$\mathcal{O}(se + s\ log\ s)$, where s is the number of distinct candidate tags. Since $e \gg s$, we can neglect all the operations on s and conclude that the complexity of *Apply_SD* is $\mathcal{O}(e)$. Since $e \leq n$, where n is the size of the Web document from which S is constructed, algorithm *Apply_SD* has time complexity $\mathcal{O}(n)$.

For example, consider the largest-fan-out sub-tree rooted at node *td* in Figure 2.5 and the three candidate tags 'hr', 'b', and 'br' as computed in Section 2.4. Algorithm *Calculate_SD* determines the standard deviation of the lengths between each pair of identical candidate tags 'hr', 'b', and 'br', respectively. In processing 'hr', *Calculate_SD* constructs $INTERVAL$ = [486, 637, 710], calculates the summation of all the elements in $INTERVAL$, which is $sum = 1833$, and computes the average of all the elements in $INTERVAL$, which is $ave = 1833/3 = 611$. Thus, $SD(hr)$ is $(((486 - 611)^2 + (637 - 611)^2 + (710 - 611)^2)/3)^{1/2} \cong 93.28$, whereas $SD(b)$ and $SD(br)$ are 200.78 and 260.33, respectively. Since $SD(hr) < SD(b) < SD(br)$, CL = [(hr, 1), (b, 2), (br, 3)] is computed by algorithm *Apply_SD*.

3.4 The Identifiable "Separator" Tags Heuristic

For documents with multiple records, there tends to be a few tags that consistently separate these records. By looking at these documents and keeping track of the separator tags and how often these separator tags are used, we can create an ordered list of the most commonly used tags that separate records of interest in Web documents.

To construct an identifiable-"separator"-tags list (called IST) that contains the most commonly used tags that separate records of interest in Web documents, we retrieved one hundred Web documents in two application areas (obituaries and car advertisements applications) from ten different Web sites, manually determined the record separators of these documents, and included them in IST. The tags in IST are ordered according to the occurrences of the tags being the separators. The tag

Algorithm *Apply_SD* /* Orders candidate tags according to their standard deviations */

Input: the largest-fan-out sub-tree S and TL, an array of ordered pairs of the form (STR, NUM), where STR is the name of a candidate tag and NUM is the number of appearance of STR in the child nodes of the root node in S /* S and TL are generated by algorithms $Find\_Largest\_Fanout$ and $Pre\text{-}Process\_Tags$, respectively */

Output: the candidate list CL that contains all the candidate tags and their rankings that are determined by their standard deviations

Begin /* Algorithm */

1. Initialize $n := 0$ and $L := []$ /* n denotes the size of CL and L is an array of ordered pairs of the form (STR, NUM) */
2. WHILE $n < |TL|$ DO /* $|TL|$ denotes the size of TL */
2.1. Set $sd := Calculate\_SD(S, TL[n].STR)$ /* calculates the standard deviation of a candidate tag */
2.2. Set $temp := (TL[n].STR, sd)$ /* $temp$ is an ordered pair (STR, NUM), and $TL[n].STR$ is the name of a candidate tag */
2.3. For each element in L, $0 \leq j < i < k < n$ and $j + 1 = i = k - 1$, if $L[j].NUM \leq temp.NUM \leq L[k].NUM$, then $L :=$ insert(L, $temp$, i) /* inserts $temp$ into the i-th position of L */
2.4. Set $n := n + 1$

   END-WHILE
3. Initialize $r := 0$ /* r is the counter of the rankings */
4. FOR $m := 0$ TO $n - 1$ DO /* determines the ranking of each tag */
4.1. Set $CL[m].STR := L[m].STR$ and $r := r + 1$
4.2. IF $m > 0$ AND $L[m].NUM = L[m-1].NUM$

   THEN $CL[m].NUM := CL[m-1].NUM$ /* assigns the same ranking to the two tags with the same standard deviation */

   ELSE $CL[m].NUM := r$ /* assigns ranking r to the candidate tag $CL[m].STR$ */

   END-IF

   END-FOR
5. Return CL

End /* Algorithm */

Algorithm *Calculate_SD* /* Calculates the standard deviation of tag T */

Input: the largest-fan-out sub-tree S and T, where T is the name of a candidate tag

Output: sd, the standard deviation of the lengths of plain text between each pair of consecutive Ts

Begin /* Algorithm */

1. Initialize $sum := 0$, $n := 0$, and $INTERVAL := []$, where $INTERVAL$ is an array of integers /* An integer in $INTERVAL$ denotes the length of the plain text between a pair of consecutive, identical candidate tags, and sum is the sum of $INTERVAL$ */
   /* Builds $INTERVAL$ */
2. Let p be the position of the first appearance of T in the start-tag element in a child node of the root node in S
3. WHILE $p \neq$ -1 /* If T is not found in the child nodes of S, then $p =$ -1 */
   3.1. Let q be the position of the next appearance of T in the start-tag element of a child node (which is the right sibling of the node that contains p), or -1, otherwise
   3.2. Initialize $len := 0$ /* len is the size of the plain text between a pair of identical candidate tags */
   3.3. IF $q \neq$ -1
   3.3.1. THEN FOR $j := p$ TO q - 1 DO
      Let u be the sub-tree rooted at the j-th child node of the root node in S, and set $len := len +$ size_of($Get\_Plaintext(u)$)
      /* size_of(String) is the number of characters in String */
      END-FOR
   3.3.2. Let $INTERVAL[n] := len$ and $n := n + 1$
   3.3.3. Let $sum := sum + len$
   END-IF
   3.4. Set $p := q$
   END-WHILE
   /* Calculates $SD(T)$ */
4. Set $ave := sum \ / \ n$
5. Initialize $sum_2 := 0$
6. FOR $j := 0$ TO n - 1 DO
   $sum_2 := sum_2 + (INTERVAL[j] - ave)^2$
   END-FOR
7. Set $sd := (sum_2/(n+1))^{1/2}$
8. Return sd

End /* Algorithm */

Record Separator	*Number of Appearance*
hr	75
tr	15
td	15
a	14
table	10
p	10
br	10
h4	10
h1	10
strong	10
b	6
i	5

Table 3.1: The record separators and the numbers of their appearances in 100 Web documents

that appears more often than the others as the record separator in the hundred Web documents is placed at the beginning of IST. Table 3.1 lists all the record separators and the numbers of their appearances in the hundred Web documents we examined. Note that since one document may contain more than one record separator, the total number of appearances of record separators can be greater than one hundred (i.e., the number of Web documents). If the appearances of two record separators are the same, we randomly order these two tags in our IST.

The identifiable-"separator"-tags (IT) heuristic is implemented in algorithm *Apply_IT* which takes a list of candidate tags generated by algorithm $Pre\text{-}Process\_Tags$ and IST (i.e., the list of tags as shown in Table 3.1) as input and returns the candidate list CL which contains the candidate record separators and their rankings. The algorithm compares the candidate tags with the tags in IST. If a tag D in IST is also a candidate tag, D is inserted into CL. Eventually, the tags in CL are ordered according to their relative positions in IST. Thus, the ranking of D is determined by its position in CL. For example, If D is the first element in CL, the ranking of D is 1.

Algorithm *Apply_IT* /* Determines whether a tag in IST is a candidate tag */

Input: TL and IST, where TL is an array of ordered pairs of the form (STR, NUM), where STR is the name of a candidate tag and NUM is the number of appearance of STR in the child nodes of the largest-fan-out node and IST consists of the names of identifiable-"separator"-tags /* TL is generated by algorithm *Pre-Process_Tags* */

Output: the candidate list CL which contains the candidate tags which appear in IST and their rankings which are determined by their relative positions in IST

Begin /* Algorithm */

1. Initialize $n := 0$ /* n denotes the size of CL */
2. FOR each element e in IST (chosen according to the left-to-right order) DO
   IF $e = TL[j].STR$, $\forall j$, $0 \leq j \leq |IST|$
   THEN Let $CL[n].STR := e$, $CL[n].NUM := n + 1$, and $n := n + 1$
   /* $CL[n].STR$ is the name of the candidate tag, whereas $CL[n].NUM$ is the ranking of $CL[n].STR$ */
   END-FOR
3. Return CL

End /* Algorithm */

Since *Apply_IT* compares each tags in IST with each candidate tag, the complexity of *Apply_IT* is $\mathcal{O}(sl)$, where s is the number of distinct candidate tags and l is the number of tags in IST. Since both the number of candidate tags s and the length of our tag list l are small compared to the size of a document, the cost of this operation is negligible.

For example, consider the Web document in Figure 2.1, tags 'hr', 'b', and 'br' are the candidate tags computed in Section 2.4. IST = [hr, tr, td, a, table, p, br, h4, h1, strong, b, i], as shown in Table 3.1. Since candidate tags 'hr', 'b', and 'br' appear in IST and since 'hr' appears in front of 'br' and 'br' in turn appears in front of 'b' in IST, algorithm *Apply_IT* simply returns CL = [(hr, 1), (br, 2), (b, 3)].

3.5 The Highest-Count Tags Heuristic

For a Web document D that contains multiple records of interest, the tag with the most appearance may be the record separator of D. In the highest-count tag (HT) heuristic, we construct a candidate list CL that contains the names of the distinct candidate tags and their rankings. The rankings of candidate tags in CL are determined by the number of appearance of each candidate tag in the child nodes of the largest-fan-out node constructed from D. The candidate tag with the largest number of appearance is ranked as 1 and placed at the beginning of CL. For example, consider the Web document in Figure 2.1 and TL = [(hr, 4), (b, 8), (br, 5)] as computed in Section 2.4. (The second element of each ordered pair is the number of appearance of its corresponding first element.) Since $8 > 5 > 4$, CL = [(b, 1), (br, 2), (hr, 3)].

The HT heuristic is implemented in algorithm *Apply_HT*. Suppose the number of distinct candidate tags is s. The candidate tags are sorted according to their numbers of appearances in the child nodes of the largest-fan-out node in a tag tree T. The complexity of sorting these candidate tags is $\mathcal{O}(s\ log\ s)$. Also, the complexity of assigning a ranking to each distinct candidate tag is $\mathcal{O}(s)$. Thus, the complexity of *Apply_HT* is $\mathcal{O}(s\ log\ s+s)$, which is $\mathcal{O}(s\ log\ s)$. Since $s \ll n$, where n is the length of the Web document from which T is constructed, we consider the cost of this operation to be negligible.

Algorithm *Apply_HT* /* Orders the candidate tags so that the candidate tag with the most appearance in TL is at the beginning of the candidate list */

Input: TL, an array of ordered pairs of the form (STR, NUM), where STR is the name of a candidate tag and NUM is the number of appearance of STR in the child nodes of the largest-fan-out node of a tag tree /* TL is generated by algorithm *Pre-Process_Tags* */

Output: the candidate list CL that contains all the candidate tags and their rankings that are determined by the second elements of the ordered pairs in TL

Begin /* Algorithm */

1. Set TL' := sort (TL) so that for each j and k such that $0 \leq j < k < |TL|$, $TL'[j].NUM \geq TL'[k].NUM$ /* the tag with the most appearance is placed at the beginning of the list */
2. Initialize $r := 0$ /* r is the counter of the rankings */
3. FOR $m := 0$ TO $|TL|$ - 1 DO /* determines the rankings of each candidate tag */

3.1. Set $CL[m].STR := TL'[m].STR$ and $r := r + 1$

3.2. IF $m > 0$ AND $TL'[m].NUM = TL'[m-1].NUM$
THEN $CL[m].NUM := CL[m-1].NUM$
ELSE $CL[m].NUM := r$
END-IF

END-FOR

4. Return CL

End /* Algorithm */

Chapter 4

The Combined Heuristics

Each individual heuristic presented in Chapter 3 is independent of the others and works well only for some particular Web documents. We therefore consider combining two or more of these individual heuristics to improve our chances of locating the correct record separator in a Web document. To determine the best combination of the five individual heuristics, we adopt *Stanford certainty theory* to help us make the decision.

In Section 4.1 we explain our adaptation of Stanford certainty theory. As will be evident in this section, we will need to have certainty factors for candidate tags determined by each of our individual heuristics (or certainty factors for short). To obtain these certainty factors, we conducted some initial experiments. In Section 4.2 we describe these initial experiments and how we used them to obtain the certainty factors. Given these certainty factors, we present in Section 4.3 the process for selecting our combined heuristic. Finally, we present in Section 4.4 our overall heuristic algorithm for discovering record separators in Web documents that contain multiple records.

4.1 Certainty Measure

Stanford certainty theory defines a confidence measure and generates some simple rules for combining independent evidence. The *certainty factor* associated with evidence E for some observation B, denoted $CF(E : B)$, is the probability with which E supports B. For example, if the observation is that a candidate tag $<A>$ is the record separator of a Web document D and $CF(<A>) = 70\%$, then $<A>$ has a 70% chance to be the record separator of D. If two independent evidence support the same observation B (e.g., two individual heuristics applied on a Web document D support the same result that a particular candidate tag is the record separator of D), Stanford certainty theory gives the following rule to combine these two evidence for B. Suppose $CF(E_1 : B)$ is the certainty factor associated with evidence E_1 for some observation B and $CF(E_2 : B)$ is the certainty factor associated with evidence E_2 for the same observation B, then the new certainty factor of B, called the *compound certainty factor* of B, is calculated by: $CF(E_1 : B) + CF(E_2 : B)$ - $CF(E_1 : B) \times CF(E_2 : B)$. By using this rule repeatedly, it is possible to combine the results of evidence from any number of independent events that are used for determining B. For example, if the certainty factors from three different heuristics are 88%, 74%, and 66% that a tag $<T>$ is the record separator in a Web document, then the compound certainty factor for $<T>$ is 98.93%. (The compound certainty factor is computed using Stanford certain theory on these three certainty factors as 88% + 74% + 66% - 88% $\times$ 74% - 88% $\times$ 66% - 74% $\times$ 66% + 88% $\times$ 74% $\times$ 66% = 98.93%.)

4.2 Initial Experiments

To determine the certainty factors for the five individual heuristics, we considered two application areas: obituaries and car advertisements. Figures 3.1 and 4.1 give

the object-relationship model instances for the obituaries and car advertisements in graphical form, respectively. To achieve geographical diversity (and thus hopefully a reasonable sampling of different kinds of Web documents), we chose ten on-line newspaper sites[1] (listed in Table 4.1) located in different regions of the United States. For each application, we retrieved five Web documents from each site. Thus, 100 experimental Web documents were examined. After scanning through these documents, we manually located the correct record separators in each of these 100 documents. (Note that a Web document may contain more than one record separator, and the same record separator may not be used in all the Web documents retrieved from the same site.) We then applied each individual heuristic H on each experimental Web document and compared the record separator retrieved by H with the manually determined record separators.

Tables 4.2 and 4.3 give the results for obituaries and car advertisements, respectively. The first row of Table 4.2 shows that 83% of the time the OM heuristic ranked a correct record separator of an experimental Web document as its first choice and 17% of the time the OM heuristic ranked a correct record separator as its second choice. Similarly, for the other heuristics, we calculated the percentage of Web documents in which a correct record separator was the first, second, third, or fourth choice of the ranking obtained from each of the individual heuristics. In these initial experiments, a correct record separator was always among the four highest ranked choices for all the individual heuristics.

By comparing the percentages of the two application areas in Tables 4.2 and 4.3, we can see that the results are reasonably consistent in both applications. We obtained

[1] In the process of choosing the sites for our initial experiments, we examined 14 different sites. At two sites their records always resided in more than one Web document and were connected by hypertext links. At another site a tag in a Web document was used for all the purposes (such as line break, paragraph break, and record separator). At yet another site two different record separators alternatively appeared at different record boundaries. Web documents located at these four sites did not fit our assumptions and thus were eliminated from consideration in our experiments.

On-line Newspaper	*URL*
The Salt Lake Tribune	http://www.sltrib.com
The Arizona Daily Star	http://www.azstarnet.com
The Houston Chronicle	http://www.chron.com
The San Francisco Chronicle	http://www.sfgate.com
The Seattle Times	http://www.seatimes.com
GoCincinnati.com	http://classifinder.gocinci.net/
The Standard Times	http://www.s-t.com/
The Detroit Newspapers	http://www.dnps.com
The Connecticut Post	http://www.connpost.com
Access Atlanta	http://www.accessatlanta.com

Table 4.1: On-line newspapers chosen for initial experiments

Heuristic Approach\Ranking	*1*	*2*	*3*	*4*
OM	83%	17%	0%	0%
RP	83%	7%	10%	0%
SD	59%	27%	14%	0%
IT	92%	8%	0%	0%
HT	58%	23%	17%	2%

Table 4.2: Experimental results for obituaries application

Heuristic Approach\Ranking	*1*	*2*	*3*	*4*
OM	86%	8%	4%	2%
RP	72%	18%	8%	2%
SD	72%	18%	10%	0%
IT	100%	0%	0%	0%
HT	40%	42%	16%	2%

Table 4.3: Experimental results for car advertisements application

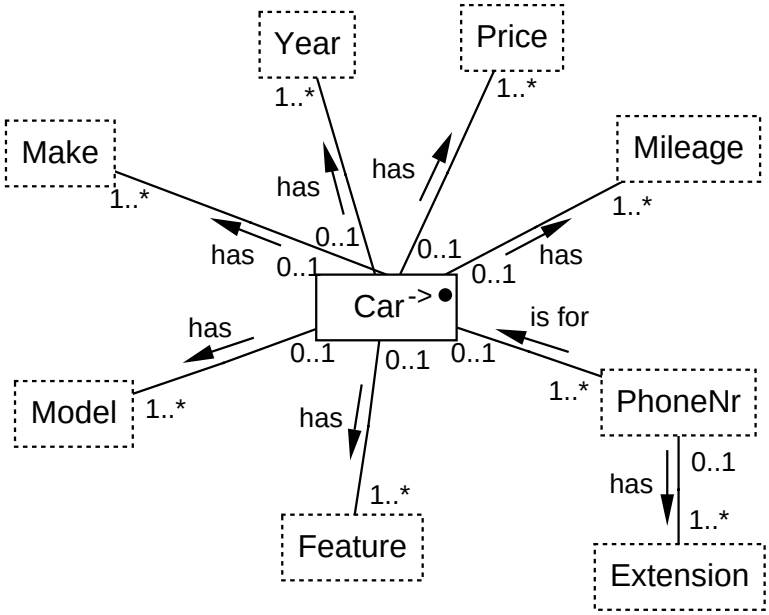

Figure 4.1: Graphical car advertisements ontology

Heuristic Approach\Ranking	*1*	*2*	*3*	*4*
OM	84.50%	12.50%	2.00%	1.00%
RP	77.50%	12.50%	9.00%	1.00%
SD	65.50%	22.50%	12.00%	0.00%
IT	96.00%	4.00%	0.00%	0.00%
HT	49.00%	32.50%	16.50%	2.00%

Table 4.4: Certainty factors, as selected by our initial experiments

our certainty factors for candidate tags in each individual heuristic by averaging the percentages in Tables 4.2 and 4.3. Table 4.4 shows the resulting certainty factors. This table asserts that the highest ranking candidate tag chosen by the OM heuristic has a certainty factor of 84.50%, that the second highest ranking candidate tag has a certainty factor of 12.50%, and so on for the OM heuristic and also for all other heuristics.

4.3 The Combined Heuristic

For our combined heuristic we had the choice of any combination of two, three, four, or all five of the individual heuristics. It might seem that choosing the top two or three individual heuristics and ignoring the rest might produce the best results. Because we did not know what combination to choose, we continued with our initial experiments and tried all combinations on the same 100 Web documents as mentioned in Section 4.2. There are twenty-six (i.e., $\sum_{i=0}^{5} C(5,i) - 6 = 26$) possible combinations. (Note that six of the possible combinations are excluded because we cannot have none and we already have the results for the five individual heuristics.)

For each combination, we calculated the compound certainty factor for each candidate tag in our experimental documents. In each experimental document, the candidate tag with the highest compound certainty factor was chosen as the record separator. We then determined the success rate of each combination on our experimental Web documents. In order to calculate the success rate of a combination on the experimental documents, we need to first obtain the success of a combined heuristic on a particular document. If there are X tags that have the highest compound certainty factors and only Y of these X tags are correct record separators in a Web document D, then the *success* for D, denoted $sc(D)$, is Y/X (i.e., there is $Y/X\%$ chance that the correct record separator in D is chosen). If all the tags that have the highest certainty factors are record separators in D (i.e., $Y = X$), then $sc(D)$ is 1 (i.e., the correct record separator in D is always chosen by a combined heuristic). If none of the tags that has the highest certainty factors is a record separator in D (i.e., $Y = 0$), then $sc(D)$ is 0 (i.e., none of the correct record separators in D is chosen by a combined heuristic). The *success rate* for a combination on n Web documents is $(\sum_{i=1}^{n}(sc(D_i))/n)$, where D_i is the i-th experimental Web document. Table 4.5 shows

Compound Heuristic	*Success Rate*	*Compound Heuristic*	*Success Rate*
OR	85.83%	OSI	95.00%
OS	88.00%	OSH	87.50%
OI	95.00%	OIH	95.00%
OH	79.00%	RSI	95.00%
RS	79.50%	RSH	85.50%
RI	95.00%	RIH	95.00%
RH	76.33%	SIH	95.00%
SI	95.00%	ORSI	100.00%
SH	69.50%	ORSH	82.50%
IH	95.00%	ORIH	100.00%
ORS	81.50%	OSIH	95.00%
ORI	93.33%	RSIH	100.00%
ORH	84.83%	ORSIH	100.00%

Table 4.5: Experimental results for all the combined heuristics

the success rates for all combinations. Note that in Table 4.5 we use O, R, S, I, and H to represent the OM, RP, SD, IT, and HT heuristics, respectively. For example, OR denotes the OM and RP combination.

By considering the success rates in Table 4.5, we realize that all the combinations that include IT have high success rates (over 90%). This is not surprising since IT, by itself, was the best in our initial experiments as Tables 4.2 and 4.3 show. We also see, however, that ORSI, ORIH, RSIH, and ORSIH all have 100% success rate for our experimental documents (i.e., in finding a correct record separator in each of the 100 experimental documents). In deciding among these four best choices, we observed that any one of them could be chosen as our combined heuristic. Since all five individual heuristics are independent and since they may all help find the correct record separator in a Web document, we decided to choose ORSIH, which includes all five individual heuristics, as the combined heuristic in our record-boundary discovery approach.

Algorithm *Apply_CH* describes the process of implementing ORSIH. For each candidate tag C in the largest-fan-out sub-tree, *Apply_CH* applies Stanford certainty theory to the results of the five individual heuristics using the certainty factors as shown in Table 4.4 and produces the compound certainty factor for C. Note that the OM, RP, or IT heuristics may not supply a result (i.e., the returned result is *null*). *Apply_CH* returns a candidate list that contains all the candidate tags and their compound certainty factors. The complexity of *Apply_CH* is $\mathcal{O}(s)$, where s is the number of distinct candidate tags.

4.4 The Record-Boundary Discovery Algorithm

We present our record-boundary discovery approach in algorithm *Discover_Record-boundary*.

For example, consider the Web document D in Figure 2.1. The results of applying the five individual heuristics are as follows:

$$OML = [(\text{hr}, 1), (\text{br}, 2), (\text{b}, 3)]$$
$$RPL = [(\text{hr}, 1), (\text{br}, 2), (\text{b}, 3)]$$
$$SDL = [(\text{hr}, 1), (\text{b}, 2), (\text{br}, 3)]$$
$$ITL = [(\text{hr}, 1), (\text{br}, 2), (\text{b}, 3)]$$
$$HTL = [(\text{b}, 1), (\text{br}, 2), (\text{hr}, 3)]$$

Applying algorithm *Apply_CH* to the results of the five individual heuristics (shown above) yields $CL = [(\text{hr}, 99.96\%), (\text{b}, 64.75\%), (\text{br}, 56.34\%)]$. Thus, 'hr' is chosen as the record separator of D since 'hr' has the highest compound certainty factor (99.96%) among all the three candidate tags.

We argued earlier that the time complexity of constructing the tag tree T of a Web document D is $\mathcal{O}(n)$, where n is the length of D, which is the time complexity of

Algorithm *Apply_CH*

Input: CFT, and six arrays TL, OML, RPL, SDL, ITL, and HTL of ordered pairs of the form (STR, NUM), which are generated by algorithms *Pre-Process_Tag*, *Apply_OM*, *Apply_RP*, *Apply_SD*, *Apply_IT*, and *Apply_HT*, respectively, for a Web document D. /* CFT is the certainty factors table as shown in Table 4.4, and STR in the last five input is a candidate tag name and NUM is the ranking of STR computed by the corresponding individual heuristic */

Output: CL, an array of ordered pairs of the form (STR, NUM), where STR is the name of a candidate tag and NUM is the compound certainty factor of STR

Begin /* Algorithm */

1. Initialize $CL := []$
2. FOR $j := 0$ TO $|TL|$ - 1 DO
2.1. Set $a := 0$, $b := 0$, $c := 0$, $d := 0$, and $e := 0$ /* a, b, c, d, and e are the certainty factors for $TL[j].STR$, each determined by its corresponding individual heuristic */
/* Find $TL[j].STR$ in each of the candidate lists generated by an individual heuristic and obtain the certainty factor from each individual heuristic */
2.2. IF $OML \neq null$ /* the OM heuristic provides an answer for D */ AND $TL[j].STR = OML[k].STR$, $\forall k$, $0 \leq k < |OML|$
THEN $a := CFT[0][k]$ /* $CFT[0]$ is the first row in Table 4.4 */
END-IF
2.3. IF $RPL \neq null$ /* the RP heuristic provides an answer for D */ AND $TL[j].STR = RPL[k].STR$, $\forall k$, $0 \leq k < |RPL|$
THEN $b := CFT[1][k]$ /* $CFT[1]$ is the second row in Table 4.4 */
END-IF
2.4. IF $TL[j].STR = SDL[k].STR$, $\forall k$, $0 \leq k < |SDL|$
THEN $c := CFT[2][k]$ /* $CFT[2]$ is the third row in Table 4.4 */
END-IF
2.5. IF $ITL \neq null$ /* the IT heuristic provides an answer for D */ AND $TL[j].STR = ITL[k].STR$, $\forall k$, $0 \leq k < |ITL|$
THEN $d := CFT[3][k]$ /* $CFT[3]$ is the fourth row in Table 4.4 */
END-IF
2.6. IF $TL[j].STR = HTL[k].STR$, $\forall k$, $0 \leq k < |HTL|$
THEN $e := CFT[4][k]$ /*$CFT[4]$ is the last row in Table 4.4 */
END-IF

/* Apply Stanford certainty theory to calculate the compound certainty factor for $TL[j].STR$ */

2.7. Set $cf := a + b + c + d + e$ - $a \times b$ - $a \times c$ - $a \times d$ - $a \times e$ - $b \times c$ - $b \times d$ - $b \times e$ - $c \times d$ - $c \times e$ - $d \times e + a \times b \times c + a \times b \times d + a \times b \times e + a \times c \times d + a \times c \times e + a \times d \times e + b \times c \times d + b \times c \times e + b \times d \times e + c \times d \times e$ - $a \times b \times c \times d$ - $a \times b \times c \times e$ - $a \times b \times d \times e$ - $a \times c \times d \times e$ - $b \times c \times d \times e + a \times b \times c \times d \times e$

/* Insert $TL[j].STR$ and cf into CL */

2.8. Set $temp := (TL[j].STR, cf)$ /* $temp$ is an ordered pair of the form (STR, NUM) */

2.9. Set $CL :=$ insert$(CL, temp)$ /* inserts $temp$ into CL */

END-FOR

3. return CL

End /* Algorithm */

Algorithm *Discover_Record-boundary*

Input: A Web document D

Output: The consensus record separator of D

Begin /* Algorithm */

1. call algorithm *Construct_Tree* (in Section 2.2) to create the tag tree T of D
2. call algorithm *Find_Largest_Fanout* (in Section 2.3) to locate the largest-fan -out sub-tree HF in T
3. call algorithm *Pre-Process_Tags* (in Section 2.4) to extract the set of candidate tags TL from HF
4. call algorithms *Apply_OM*, *Apply_RP*, *Apply_SD*, *Apply_IT*, and *Apply_HT* (in Chapter 3) to apply the five individual heuristics OM, RP, SD, IT, and HT using TL and other inputs as detailed in each algorithm, respectively
5. call algorithm *Apply_CH* to produce the compound certainty factor for each candidate tag
6. choose the candidate tag with the highest compound certainty factor computed in Step 5 as the record separator of D

End /* Algorithm */

Step 1 in algorithm *Discover_Record-boundary*. Locating the largest-fan-out sub-tree of T in Step 2 and creating TL in Step 3 take a constant amount of time. Applying each individual heuristic in Step 4 takes at most $\mathcal{O}(n)$ time, with the understanding that D is a Web document found in practice and that the regular-expression matching for the OM heuristic has already been done for the overall data-extraction process. Computing the compound certainty factor for each of the candidate tags in Step 5 using Stanford certainty theory is $\mathcal{O}(s)$, where s is the number of candidate tags, as is choosing the candidate tag with the highest compound certainty factor. Hence, the entire process for discovering the record separator of D is $\mathcal{O}(n)$ for practical cases within the context of the larger data-extraction problem.

Chapter 5

Experimental Results

To verify the accuracy of our heuristic approach in discovering the correct record separators, we examined four sets of Web documents in four different application areas. Each set contained five Web documents, each one retrieved from a different Web site, and there were twenty Web documents all together. The twenty Web sites we chose were located in different regions of the United States. Two of the sets were documents for obituaries and car advertisements, the applications that we used in our initial experiments (see Chapter 4). However, we chose different groups of Web documents in these two application areas from entirely different sites to verify our approach (compare Table 4.1 with the site listings in Tables 5.1 and 5.2). The other two sets of Web documents were for two entirely different applications, namely computer job advertisements and university course descriptions (see the site listings in Tables 5.3 and 5.4). The ontology for these two applications are given in Figures 5.1 and 5.2, respectively.

For each of the twenty Web documents, we applied the five individual heuristics presented in Chapter 3 and ORSIH, selected as our combined heuristic as explained in Chapter 4. For the OM heuristic, we chose the record-identifying fields listed in Figure 5.3 as our ontology rules for the corresponding application. Note that in Figure 5.3 "KEYWORD(X)" denotes the keyword indicator associated with the set of

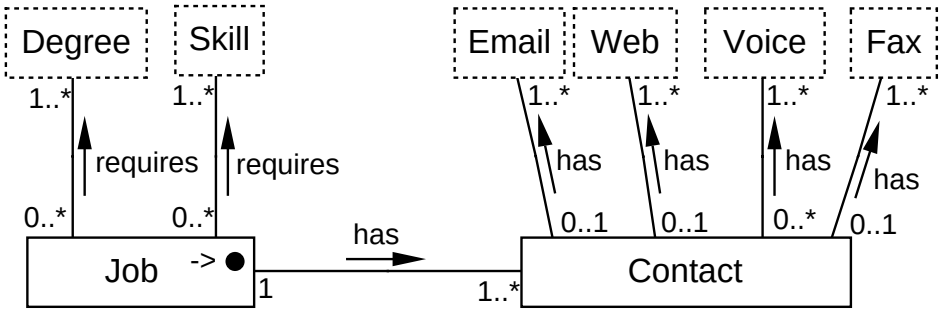

Figure 5.1: Graphical computer job advertisements ontology

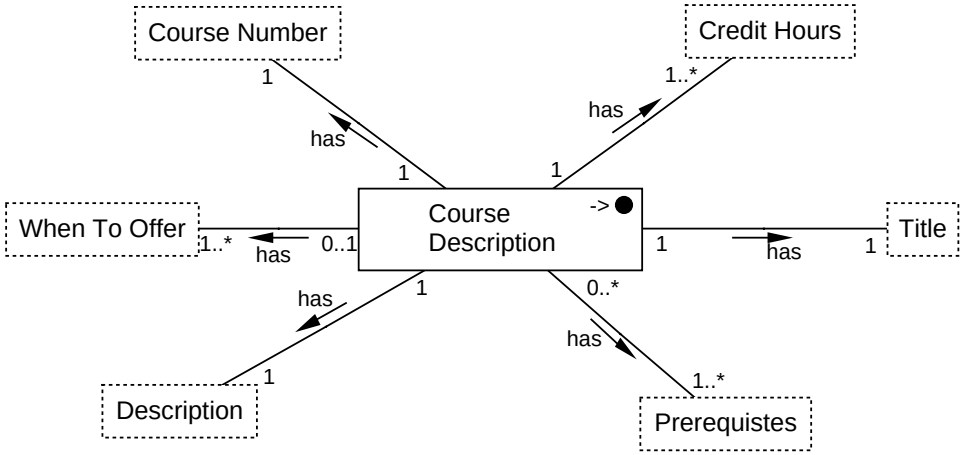

Figure 5.2: Graphical university course descriptions ontology

Obituaries
    KEYWORD(Internment)
    KEYWORD(Funeral)
    KEYWORD(BirthDate)
    KEYWORD(DeathDate)
Car advertisements
    Make
    Model
    KEYWORD(Mileage)
Computer job advertisements
    KEYWORD(Voice)
    KEYWORD(Fax)
    Voice
University course descriptions
    KEYWORD(CreditHour)
    CourseNumber
    KEYWORD(Prerequisite)

Figure 5.3: The ontology rules for the OM heuristic

objects X in the ontology rules (see Figure 3.2 for the obituaries application), whereas the identifiable value indicator associated with the set of objects Y in the ontology rules is denoted by "Y" in Figure 5.3 (e.g., *buick* is an identifiable value associated with the set of objects $MAKE$ in the car advertisements application). The results of the rankings of the record separators generated by each heuristic are shown in Tables 5.1 - 5.4. The numbers in each column are the rankings of the correct record separator obtained by the heuristic. (Note that we assign the ranking to a candidate tag DT determined by a combined heuristic H based on the compound certainty factor for DT using H. A candidate tag with the highest compound certainty factor is ranked 1 in H.) For the car advertisements appeared in the *Sioux City Journal* in Table 5.2, for example, the OM heuristic ranked the correct record separator first, RP ranked it second, SD also ranked it second, IT ranked it first, HT ranked it fourth, and ORSIH ranked it first.

We also calculated the success rate for each heuristic on all experimental Web

On-line Newspaper	*URL*	*OM*	*RP*	*SD*	*IT*	*HT*	*ORSIH*
Alameda Newspaper	http://www.adone.com/alameda	1	1	1	1	1	1
Idaho State Journal	http://www.journalnet.com	1	1	2	1	2	1
Sacramento Bee	http://www.sacbee.com	1	1	1	1	1	1
Tampa Tribune	http://www.tampatrib.com	1	1	1	1	1	1
Shoals Timesdaily	http://www.timesdaily.com	1	1	1	1	2	1

Table 5.1: Test set 1 - obituaries

On-line Newspaper	*URL*	*OM*	*RP*	*SD*	*IT*	*HT*	*ORSIH*
Arkansas Democrat-Gazette	http://www.ardemgaz.com	1	1	1	1	2	1
Sioux City Journal	http://www.siouxcityjournal.com	1	2	2	1	4	1
Knoxville News	http://www.knoxnews.com	1	1	1	1	1	1
Lincoln Journal Star	http://www.nebweb.com	1	1	1	1	1	1
Reno Gazette - Journal	http://www.nevadanet.com/renogazette	3	3	1	1	3	1

Table 5.2: Test set 2 - car advertisements

On-line Newspaper	*URL*	*OM*	*RP*	*SD*	*IT*	*HT*	*ORSIH*
Baltimore Sun	http://www.sunspot.net	1	1	1	1	2	1
Dallas Morning News	http://dallasnews.com	1	1	2	1	2	1
Denver Post	http://www.denverpost.com	4	1	1	1	4	1
Indianapolis Star/News	http://www.starnews.com	1	1	1	1	1	1
Los Angeles Times	http://www.latimes.com	2	3	2	1	2	1

Table 5.3: Test set 3 - computer job advertisements

University	*URL*	*OM*	*RP*	*SD*	*IT*	*HT*	*ORSIH*
Brigham Young University	http://www.byu.edu	2	2	1	1	1	1
MIT	http://registrar.mit.edu	1	1	1	1	2	1
Kansas State University	http://www.ksu.edu	1	1	2	2	2	1
USC	http://www.usc.edu	1	1	2	1	1	1
Univ. of Texas - Austin	http://www.utexas.edu	1	2	2	1	1	1

Table 5.4: Test set 4 - university course descriptions

Heuristic Approach	*Success Rate*
OM	80%
RP	75%
SD	65%
IT	95%
HT	45%
ORSIH	100%

Table 5.5: Success rates of individual heuristics and ORSIH for experimental Web documents

documents. Table 5.5 shows the results. (The success rates for all the individual heuristics are very close to the success rates presented in Chapter 4.) We note that even though none of the five individual heuristics had a 100% success rate, the success rate for our combined heuristic (i.e., ORSIH) that we chose in Chapter 4 is 100% on the experimental Web documents used in this chapter.

Chapter 6

Concluding Remarks

We have described a heuristic approach to discovering record boundaries in semistructured or unstructured Web documents which contain multiple records of interest separated by one or more tags. In our approach, we (1) defined a tag tree T to capture the structure of tags nested in a raw Web document, (2) located the sub-tree containing the records of interest by searching for the largest-fan-out sub-tree in T, (3) identified candidate tags within the sub-tree, (4) applied five individual heuristics (ontology-matching, repeating-tag pattern, standard deviation, identifiable "separator" tags, and highest-count tags) to rank candidate tags, and (5) combined the results of the five individual heuristics to select a consensus record separator by adopting Stanford certainty theory. For practical cases and in the context of the overall data-extraction process, the record-boundary-discovery process is $\mathcal{O}(n)$, where n is the size of a Web document.

We applied this record-boundary-discovery approach in four different application areas (obituaries, car advertisement, job advertisement, and university course description) using Web documents obtained from twenty different Web sites. The experiments we conducted showed that this approach uniformly attained an accuracy of 100%.

Although we have accomplished our goal by showing that our approach to discover

record boundaries in a Web document is promising, much remains to be done. As for future work, we note that our approach assumes that each Web document we process (1) contains multiple records and (2) contains at least one record separator. In this thesis, we do not verify these assumptions on an input Web document, nor do we solve similar document classification problems such as determining if a record spans over multiple Web documents or if a Web document only contains a single record. Furthermore, we have done all our work with HTML documents; however, we believe that most of our work should carry over directly to other DTDs, such as XML. All these issues will be of interest in our further research.

Bibliography

[Abi97] S. Abiteboul. Querying semi-structured data. In *Proceedings of International Conference on Database Theory(ICDT)*, pages 1–18, Delphi, Greece, January 1997.

[Ade98] B. Adelberg. Nodose - a tool for semi-automatically extracting structured and semistructured data from text documents. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 283–294, Seattle, Washington, June 1998.

[AK97a] N. Ashish and C. Knoblock. Semi-automatic wrapper generation for internet information sources. In *Proceedings of the CoopIS'97*, 1997.

[AK97b] N. Ashish and C. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, December 1997.

[AM97] P. Atzeni and G. Mecca. Cut and paste. In *Proceedings of the 16th ACM PODS*, pages 144–153, May 1997.

[Ape94] P. M. G. Apers. Identifying internet-related database research. In *Proceedings of the 2nd International East-West Database Workshop*, pages 183–193, Klagenfurt, 1994. Springer-Verlag.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory (ICDT)*, 1997.

[DEW97] R.B. Doorenbos, O. Etzioni, and D.S. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the First International Conference on Autonomous Agents*, pages 39–48, Marina Del Rey, California, February 1997.

[ECJ+98] D.W. Embley, D.M. Campbell, Y.S. Jiang, Y.-K. Ng, R.D. Smith, S.W. Liddle, and D.W. Quass. A conceptual-modeling approach to extracting data from the web. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER'98)*, pages 78–91, Singapore, November 1998.

[EKW92] D.W. Embley, B.D. Kurtz, and S.N. Woodfield. *Object-oriented Systems Analysis: A Model-Driven Approach.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[Emb80] D.W. Embley. Programming with data frames for everyday data items. In *Proceedings of the 1980 National Computer Conference*, pages 301–305, Anaheim, California, May 1980.

[GHR97] A. Gupta, V. Harinarayan, and A. Rajaraman. Virtual database technology. *SIGMOD Record*, 26(4):57–61, December 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*, Tucson, Arizona, May 1997.

[KWD97] N. Kushmerick, D.S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the 1997 International Joint Conference on Artificial Intelligence*, pages 729–735, 1997.

[LS98] G.F. Luger and W.A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving, Third Edition.* Addison Wesley Longman, Inc., 1998.

[MMK98] I. Muslea, S. Minton, and C. Knoblock. Stakler: Learning extraction rules for semistructured, web-based information sources. In *Proceedings of AAAI'98: Workshop on AI and Information Integration*, Madison, Wisconsin, July 1998.

[Sod97] S. Soderland. Learning to extract text-based information from the world wide web. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 251–254, Newport Beach, California, August 1997.

[Wit89] Robert S. Witte. *Statistics.* Holt, Rinehart and Winston, Inc., New York, 1989.

[WWW] Homepage for BYU data extraction research group. URL: http://osm7.cs. byu.edu/deg/index.html.