Detecting and Representing Relevant Web Deltas in WHOWEDA

SOURAV S BHOWMICK[1] SANJAY MADRIA[2] WEE KEONG NG[1]

College of Engineering[1],
School of Computer Engineering,
Nanyang Technological University, Singapore 639798
{assourav,awkng}@ntu.edu.sg

Department of Computer Science[2],
University of Missouri-Rolla,
Rolla, MO 65409
madrias@umr.edu

Abstract

In this paper, we present a mechanism for detecting and representing changes given the old and new versions of a set of interlinked Web documents, retrieved in response to a user's query. In particular, we show how to detect and represent *web deltas*, i.e., changes in the Web documents that are relevant to a user's query in the context of our *web warehousing* system called WHOWEDA (*W*are*h*ouse *o*f *We*b *Da*ta). In WHOWEDA, Web information are materialized views stored in *web tables* in the form of *web tuples*. These web tuples, represented as directed graphs, can be manipulated using a set of *web algebraic operators*. In this paper, we present a mechanism to detect *relevant* web deltas using web algebraic operators such as the *web join* and the *outer web join*. Web join is used to detect *identical documents* residing in two web tables whereas outer web join, a derivative of web join is used to identify *dangling web tuples*. We show how to represent these changes using *delta web tables*. We develop formal algorithms for the generation of delta web tables identifying web documents which have been added, deleted or modified since the last query.

Keywords: web deltas, web warehouse, web join, outer web join, delta web tables, algorithm.

1 Introduction

The Web offers access to large amounts of heterogeneous information and allows this information to change at any time and in any way. These rapid and often unpredictable changes to the information create a new problem of detecting and representing changes. This is a challenging problem because the information sources in the Web are autonomous and typical database approaches to detect changes based on triggering mechanisms are not usable. Moreover, these sources typically do not keep track of historical information in a format that is accessible to the outside user [10].

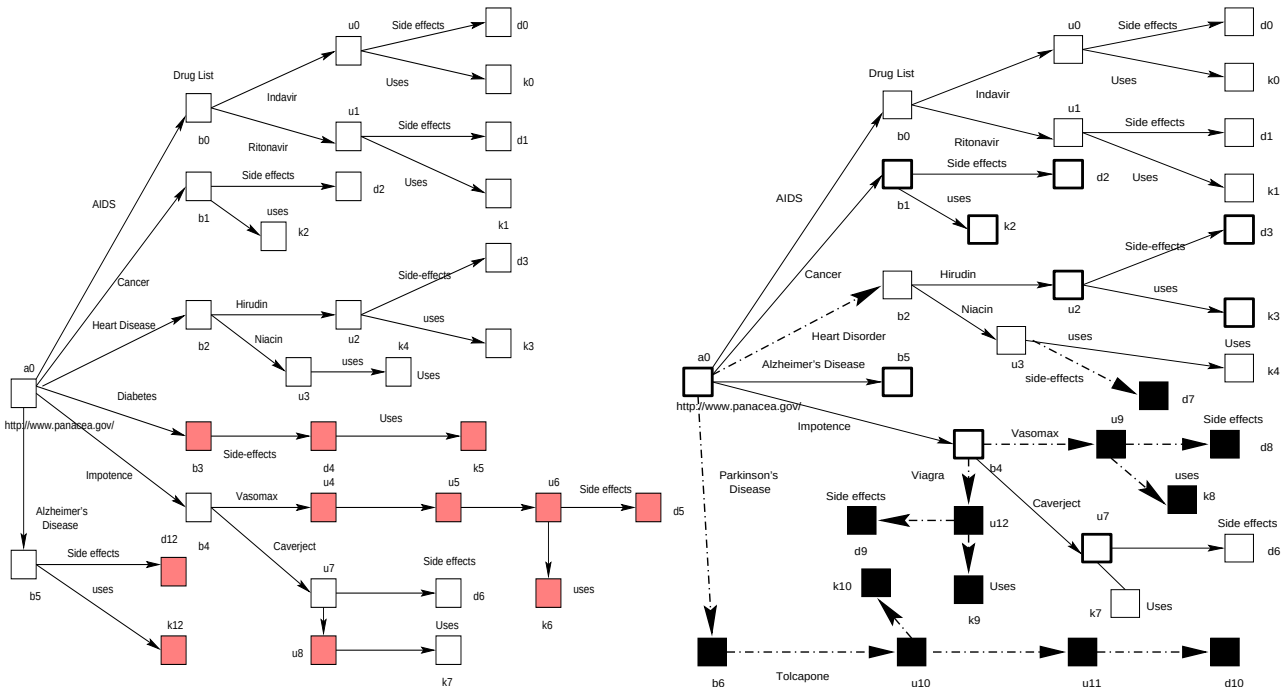
Recently, there has been increased research interest in detecting changes in structured and semistructured data [13, 14, 15]. In this paper, we present a mechanism for detecting and representing changes in Web documents (henceforth referred to as *web deltas*) which are relevant to a user's query using two *web algebraic operators*, i.e. *web join* and *outer web join*, in the context of our web warehousing system called WHOWEDA(*W*are*h*ouse *O*f *We*b *Da*ta) [4, 7] [1]. Such a mechanism for detection and representation of web deltas may be used by the following types of web users: (1) *Web site administrators:* By scanning the changes, administrators will be sure whether the changes are consistent with any policies for content or format without having to review the entire set of pages at the same level of detail. (2) *Customers of E-commerce Web sites:* A user may wish to monitor new products, services or auction on E-commerce Web sites. (3) *Analysts for gathering competitive intelligence:* Companies can monitor evolution of their competitors' Web sites to discover their new directions or offerings over a period of time that may influence their market positions. (4) *Developers of web mining applications:* By detecting and representing web deltas over a broad vistas of time, our system can be used as the foundation for mining information related to trends, patterns etc.. (5) *Wireless users:* The ability to download or highlight only changes instead of a complete Web page can be a very desirable feature for wireless users using handheld devices.

1.1 Motivating Example

We illustrate with an example some of the changes that may take place in the Web. We will use this as a running example in the rest of this paper.

Assume that there is a Web site at `http://www.panacea.gov/` which provides information related to drugs used for various diseases. For instance, the structure of the site as on 15th January, 2001 is shown in Figure 1(a). We can see that the Web page at `http://www.panacea.gov/` (denoted by a_0) contains a list of diseases. From this list, each link of a particular disease points to a Web page (denoted by b_0, b_1, b_2 etc. for various drugs) containing a list of drugs used for prevention of the disease. For example, the link labeled "AIDS" in the Web page at `http://www.panacea.gov/` points to a document (denoted by b_0) containing the list of drugs (i.e., "Indavir", "Ritonavir" etc.) used against AIDS. From the hyperlinks associated with each drug, one can probe further to find documents (denoted by u_0, u_1 etc.) containing a list of various issues related to a particular drug, i.e., "description", "manufacturers", "clinical pharmacology", "uses", "side-effects", "warnings" etc.. From the hyperlinks associated with each issue, one can retrieve details of these issues for a particular drug. Note that in Figure 1, we only show the links

[1] A shorter version of this paper appeared in [3].



(a) Web site on 15th January, 2001. (b) Web site on 15th February, 2001.

Figure 1: Web site at http://www.panacea.gov/.

related to "uses" and "side-effects" of drugs to simplify visualization.

Let us consider some modifications to this Web site on 15th February, 2001 as shown in Figure 1(b). The black boxes, the grey boxes and the boxes with thick boundaries in this figure (and all the figures in this paper except Figure 4) depict addition of new documents, deletion of existing documents and modification of existing documents respectively. Furthermore, the dashed dotted arrows indicate addition, deletion or modification of hyperlinks. Observe that the modification of the link structure of "Impotence". Previously, the information related to "Vasomax", a drug used against "Impotence" was provided by the Web site at `http://www.pfizer.com/` (Web pages u_4, u_5, u_6, d_5 and k_6 in Figure 1(a) belong to the Web site at `http://www.pfizer.com/`). That is, the link labeled as "Vasomax" in b_4 in Figure 1(a) was a global link. Now this information is provided locally by `http://www.panacea.gov/` and the structure of inter-linked documents are modified as shown in Figure 1(b) (documents u_9, d_8 and k_8).

Suppose on 15th January, 2001, a user wishes to find out periodically (say every 30 days) information related to side effects and uses of drugs for various diseases and also changes to this information compared to its previous version. This query requires access to previous states of the Web site and a mechanism to detect these changes automatically, features that are not supported by the Web or the existing search engines. Thus, we need a mechanism to compute and represent

changes in the context of Web data.

1.2 Overview

The work on change detection and representation reported in this paper has four key characteristics:

- **Relevant web deltas:** We focus on detecting *relevant* web deltas. In particular, our goal is to detect and represent web deltas that are relevant to a user's query, not any arbitrary web deltas.
- **Changes in inter-linked Web documents**: Our focus is on detecting and representing relevant changes given old and new versions of a set of inter-linked Web documents. In particular, we are interested in detecting those Web documents in a Web site which are added to or deleted from the site, or those documents which are no longer considered relevant to a user's query. We also want to identify a set of documents which have undergone content modification compared to their antecedent. Furthermore, we wish to determine how these modified Web documents are related to one another and with other relevant Web documents in the context of a user's query.
- **Web algebraic operators:** We present a mechanism for detecting and representing relevant web deltas using a set of *web algebraic operators*. These operators are applied on a sequence of Web data snapshots to infer changes.
- **Static Web pages:** Web documents that do not provide the last modification date, such as the output from Common Gateway Interface (CGI) scripts are not considered in this paper for change detection.

Our goal is to detect and represent changes in Web data using a set of web algebraic operators in the context of WHOWEDA, a data warehousing system for managing and manipulating relevant data extracted from the Web [4]. Informally, our web warehouse is conceived of as a collection of *web tables*. A set of *web tuples* and a set of *web schemas* [6] is called a web table. A web tuple is a directed graph consisting of a set of *nodes* and a set of *links* and satisfies a *web schema*. Nodes and links contain content, metadata and structural information associated with the Web documents and hyperlinks among the Web documents. To facilitate manipulation of Web data stored in the web tables, we have defined a set of web algebraic operators (i.e. *global web coupling*, *web join*, *web select* etc..) [3, 4, 7, 8, 22].

Specifically, a web join operator is used to combine *identical* data residing in two web tables. In web join, the web tuples from two web tables (say W_1 and W_2) containing *joinable nodes*

(nodes participating in web join operation) are *concatenated* into a single joined web tuple that can be materialized in a web table. A pair of nodes are *joinable* if they are *identical* in content. We consider two nodes or Web documents identical when they have the same URL and last modification date. Observe that based on this definition of identity of Web documents, same documents stored in mirror sites having different URL are not considered identical. The web tuples $w_a \in W_1$ and $w_b \in W_2$ are *concatenated* over the joinable nodes to create a joined web tuple. The joined web table contains such a set of joined web tuples.

The web tuples in W_1 and W_2 that do not participate in the web join operation (*dangling web tuples*) are absent from the joined web table. The *outer web join* operator, a derivative of web join, identifies these dangling web tuples in W_1 and W_2. We define two flavors of outer web join, i.e., *left* and *right* outer web join to identify the dangling web tuples from W_1 and W_2 respectively.

As Web data in our web warehouse are materialized views stored in the form of web tables, any changes to the relevant Web data are also reflected in the corresponding web tables. Consequently, in order to detect web deltas, we materialize the old and new versions of data in two web tables. Next, we create a set of web tables by manipulating these input web tables using the web join and outer web join operators. Finally, we create a set of *delta web tables* by further manipulating the joined and outer joined web tables. Delta web tables encapsulate the changes that have occurred in the Web such as addition, modification or deletion of a set of Web documents in the context of a user's query.

2 Related Work

In recent years, several tools have become available to address the problem of determining when an HTML page has changed. URL-minder [1] runs as a service on the Web itself and sends an email when a page changes. However, the need to send URLs explicitly through a form is cumbersome and may not be a feasible option when there is a large number of URLs to track.

The AT & T Internet Difference Engine (AIDE) [16] is a system that finds and displays changes to pages on the World Wide Web. A tool called *HtmlDiff* highlights changes between versions of a page, and a graphical interface to view the relationship between pages over time. *HtmlDiff* automatically compares two HTML pages and creates a "merged" page to show the differences with special HTML markups. TopBlend [11] is an HTML differencing tool implemented in Java and significantly outperforms the old HtmlDiff in the most time-consuming jobs.

AIDE also supports recursive tracking and differencing of a page and its descendants. When recursion is specified, changes to the child pages are reported separately by default. A user may

specify a number of operations in AIDE which includes registering a URL including the degree of recursion through links to other pages, view textual differences between a pair of versions, view a graph showing the structure of a page etc.

WebGUIDE (Web Graphical User Interface to a Difference Engine) [17] is another tool that supports recursive document comparison: users may explore the differences between the pages with respect to two dates. Differences between pages are computed automatically and summarized in a new HTML page, and the differences in link structure are shown via graphical representations. WebGUIDE is a combination of two tools, Ciao [12] and the AIDE. With Ciao, the high level structural differences are displayed as graphs that show the relationships between pages using colored nodes to indicate which pages have been modified. Using AIDE, the low-level textual differences are illustrated by marking changes between versions and modifying anchors to cause documents reached from that page to be annotated. WebGUIDE allows a user to issue queries for specific types of deltas.

The AIDE and WebGUIDE have certain limitations. First, we believe that specifying a set of URLs to track changes may not be feasible when there is a large number of URLs. Second, the recursion specification is restrictive. That is, it selects all the children of a specified document(s), however, in reality, a user may often be interested in only some of those links. On top of that, the user may wish to track changes of successive inter-linked documents satisfying some hyperlinked structure. Such constraints cannot be specified in AIDE. Third, AIDE displays all the changes in the documents. In reality we may not be interested in all the changes but only some of these changes. Hence, it is necessary to be able to query these changes rather than browsing them to find the relevant changes. This is extremely useful when the number of documents monitored is large.

In [25], the authors define a change detection problem for ordered trees, using insertion, deletion and label-update as the edit operations. In [14], the authors discuss a variant of change detection problem for ordered trees using subtree *moves* as an edit operation in addition to insertions, deletions and updates, and presented an efficient algorithm for solving it. They focus on the problem of detecting changes given the old and new versions of hierarchically structured data. Change detection problem for unordered trees is presented in [13]. The authors present efficient algorithms to detect changes in operations that moves an entire sub-tree of nodes and that copies an entire sub-tree. More recently in [10], a *snapshot-delta* approach has been used for representing changes in semistructured data. The authors present a simple and general model called *DOEM* for representing changes and also present a language called *Chorel* for querying

changes represented in *DOEM*. This model is founded on the OEM data model and the Lorel query language [2]. It uses *annotations* on the nodes and arcs of an OEM graph to represent changes. Intuitively, the set of annotations on a node or arc represents the history of that node or arc. An important feature of this approach is that it represents and queries changes directly as annotation on the affected area instead of representing them indirectly as the difference between database states. Furthermore, they describe the design and implementation of an application of change management called a *Query Subscription Service*(QSS). QSS can be used to notify subscribers of relevant changes in semistructured information sources.

DOEM was not specifically developed for the Web, and the model does not distinguish between graph edges that represent the connection between a document and one of its parts, and edges that represent a hyperlink from one Web document to another. We take a different approach as compared to [10, 14]. Our approach is specifically developed for finding changes to Web data. Rather than finding changes to the internal structure and content of Web documents, in this paper we focus on identifying changes to a set of hyperlinked Web documents relevant to a user's query. Our approach can easily be extended to detect and represent changes to internal structure and content of Web documents.

WebCQ system [20] is a prototype system for Web information monitoring and delivery. It provides a personalized notification of what and how Web pages of interest have been changed and personalized summarization of Web page changes. Users' update monitoring requests are modelled as *continual queries* [21] on the Web. WebCQ has the same limitations as of AIDE described earlier.

Recently, XML research community has recognized the importance of change management problem and in [15], the authors have discussed change management in the context of XML data. Their approach of change management is based on tree-comparison. However, they do not address the problem of detecting changes to hyperlinked XML documents. Moreover, we only find changes that affect user's query responses. However, in their approach, they find changes between any two given versions of XML data. Our approach can be extended for detecting and representing changes in XML data.

Finally, our approach is different from graph matching and isomorphic graph problems. A matching in a graph is a set of edges such that every vertex of the graph is on at most one edge in the set. Two graphs are isomorphic if one can label both graphs with the same labels so that every vertex has exactly the same neighbors in both graphs. Some of the related research on graph matching problems are [24, 27, 28]. In our approach, two non-isomorphic graphs may also

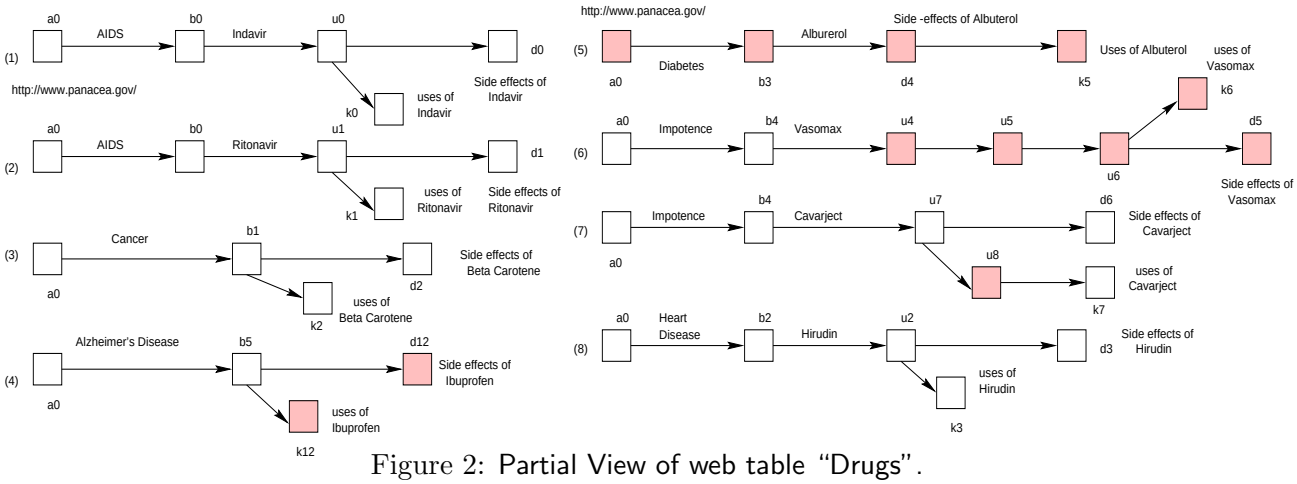


Figure 2: Partial View of web table "Drugs".

join if some of the nodes are identical. Thus, we focus more on the node content similarity rather than the structure of the graph.

3 Preliminaries

In this section, we provide the framework for our subsequent discussion on change detection.

3.1 WHOM - The Data Model of WHOWEDA

The *WareHouse Object Model* (WHOM) [4] serves as the basic data model for our web warehousing system. Informally, our web warehouse can be conceived of as a collection of *web tables*. A web table is a 3-tuple $W = \langle Z, S, T \rangle$ where Z is the web table name, S is a set of *web schemas* [6] and T is a set of *web tuples* satisfying S. A web tuple is a directed graph consisting of sets of *node* and *link objects* (hereafter, referred to as *nodes* and *links* respectively for brevity). Figures 2 and 3 are examples of web tuples in two web tables [2]. *Web schemas* are used to bind a set of web tuples in a web table and defines the structure of a set of data in the warehouse. The reader may refer to [6] for details on web schemas.

Intuitively, a *node* represents the metadata associated with a Web document and the content and structure of the document (excluding hyperlinks in the document). Specifically, it consists of two components: different metadata associated with the document (such as `URL`, `date` and `size` etc.) and a directed labeled tree to represent the content and structure of the document. Similarly, a *link* consists of a set of link meta-attribute/value pairs (such as `target URL`, `source URL` and `link type`[23]) and a *link data tree*. Link data tree is a directed labeled tree to represent

[2] Note that in all figures related to web tables, the web tuples are numbered for reference.

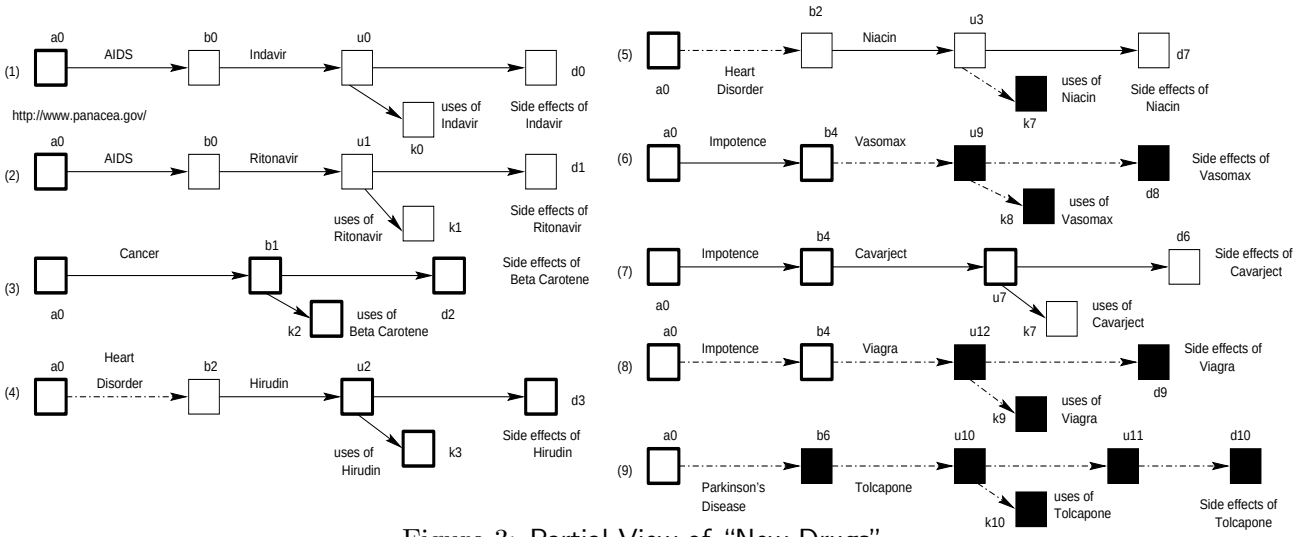


Figure 3: Partial View of "New Drugs".

the structure and content of an HTML or XML link [3]. In Figures 2 and 3, the boxes represent node objects and the arrow between two nodes represents the link object. The reader may refer to [4] for complete discussion on Web data representation.

3.2 Global Web Coupling

Global web coupling [4, 8, 22] enables a user to retrieve a set of inter-linked documents satisfying a *web query*, regardless of the locations of the documents in the Web. To initiate a global web coupling, the user specifies a web query in the form of a *coupling query* [5]. The global web coupling operator Γ takes in a coupling query G and returns a web table $W = \langle Z, S, T \rangle$ containing a set of web tuples T extracted from the WWW satisfying the query and a set of web schemas S generated from G and T. That is, $W = \Gamma(G)$. Each web tuple matches a portion of the WWW satisfying the *constraints* described in G. These constraints are imposed on the metadata, content and structure of Web documents and hyperlinks. We have omitted discussion on the coupling query here for space constraints. We assume that the sets of inter-linked documents retrieved by the given coupling query are materialized in the web tables Drugs and New Drugs respectively as shown in Figures 2 and 3. Each web tuple in these tables contains information about side effects and uses of a drug used for a particular disease. Observe that in Figure 1(a) information related to the drug "Niacin" used for Heart diseases (documents u_3 and k_4) is not materialized in Drugs as it does not satisfy the coupling query. However, it is materialized in New Drugs as it satisfies the coupling query on 15th February due to the addition of a document related to the side-effects of "Niacin" (d_7) to the Web site at `http://www.panacea.gov/`. Notice that the web

[3] We only consider simple and extended XML links.

tuples related to Diabetes and Alzheimer's disease are not materialized in New Drugs due to the removal of documents b_3, d_4, k_5, d_{12} and k_{12} from the Web site.

3.3 Storage of Web Objects

In this section, we briefly introduce various physical storage structures in WHOWEDA for storing Web objects, i.e., nodes, links, Web documents, web tables etc. We introduce three types of storage structures; *warehouse node pool*, *warehouse document pool* and *web table pool* for storing Web objects. The warehouse node pool contains distinct nodes from all web tables in our web warehouse. Each node represents a Web document stored in the warehouse document pool. The links in each Web document are stored in this pool along with the corresponding node. Furthermore, each node has an identifier called *node id*. Note that the node id of a node is different from that of another node if their URLs are different. A node may have several *versions*. In order to distinguish between several versions of a node, each version of a node is identified by a unique *version id*. Note that each node id can have a set of version ids across different web tables. A node in our web warehouse can be uniquely identified by the pair (node id, version id).

The web tables are stored in the *web table pool*. Each web table in this pool is stored in three types of structures, i.e., the *table node pool*, the *web tuple pool* and the *web schema pool*. For each distinct node and link object in a web table, we store the following attributes in the table node pool: identifier that the node and the link represent in the web schema, node id and link id, version id and URL of the node, target node id, label and link type of the link. Next, we store the web tuples of a web table in the web tuple pool. For each tuple in this pool, we only store the ids of all the nodes and links belonging to that tuple. Finally, we store the web schemas and coupling query in the web schema pool. The reader may refer to [29, 30] for detailed exposition on these storage structures.

4 Change Detection Problem

In this section, we first describe the change detection problem. Then, we identify the basic *change operations* in WHOWEDA corresponding to the changes in the Web. Finally, we show how to represent these changes in the context of our web warehouse.

4.1 Problem Definition

As changes in relevant Web data are reflected on the web tables, we can address the problem of detecting and representing changes to Web data in the context of such web tables. We first

describe the problem informally using the example in Section 1.1. Recall from Section 1.1, a user wishes to find a list of drugs for various diseases, their side effects and uses starting from the Web site at `http://www.panacea.gov/`. The user specifies a polling coupling query with polling times t_1 = 15th January, 2001, t_2 = 15th February, 2001. At polling time t_1, the global web coupling operation retrieves a set of inter-linked documents and materializes them in the form of a web table called Drugs as depicted in Figure 2.

Before polling time t_2, the Web site at `http://www.panacea.gov/` is modified as depicted in Section 1.1. Therefore, at t_2, the result New Drugs (Figure 3) of the polling coupling query contains the relevant changes that have occurred between time t_1 and t_2. Given two such web tables Drugs and New Drugs containing the snapshots of two versions of relevant Web data, the problem of change detection is to find the set of web tuples containing nodes which are inserted into or deleted from Drugs or those nodes which are modified in Drugs to transform it into New Drugs. Note that these web tuples will reflect the changes to the Web site that are relevant to the user.

4.2 Types of Changes

Changes to the web tables are reflected on the individual web tuples in WHOWEDA. Consequently, the different types of change operations in WHOWEDA can be defined in terms of the following:

- **Insert Node**: Intuitively, the operation **Insert Node** creates a set of nodes N in a web tuple in the web table W. The nodes must be new, i.e., N must not occur in W before. Note that N can be a new web tuple added to W or it can be a set of nodes inserted into an existing web tuple in W.
- **Delete Node**: This operation is the inverse of the **Insert Node** operation. It removes a set of nodes from W.
- **Update Node**: The operation **Update Node** modifies the contents of the nodes in a web tuple. By content modification of a node, we mean the textual contents or structure of the node may change or the attributes of the links embedded in the node may change.
- **Insert Link**: Intuitively, the operation **Insert Link** creates a set of links L in a web tuple in the web table W. The links must be new, i.e., L must not occur in the web tuple before. Observe that in a web table a new link can occur in two ways. First, a new link may connect an existing node to a new node. In this case, the **Insert Link** results in a **Insert Node** operation. Second, a new link may connect to existing nodes in a web tuple. In this case, the **Insert Link** operation is equivalent to the **Update Node** operation as the source node

of the link has to be modified in order to incorporate the new link. So we can express the **Insert Link** operation by the **Insert Node** or **Update Node** operation.

- **Delete Link**: It removes a set of links from W. Similar to **Insert Link**, in a web table deletion of a link may occur based on two cases. First, removal of a link may remove the only link to an existing node. In this case, it is equivalent to the **Delete Node** operation. Second, a link may be deleted between two nodes which are connected by more than one link. In this case, this operation is essentially the **Update Node** operation. Therefore, we can express the **Delete Link** operation by the **Delete Node** or **Update Node** operation.
- **Update Link**: This operation involves modification of the anchor of a link. Hence, it is essentially an **Update Node** operation.

4.3 Representing Changes

We define a structure called the *delta web table* for representing web deltas. Delta web tables encapsulate the relevant changes that have occurred in the Web with respect to a user's query. We define the following three types of delta web tables to represent the above types of change operations:

- **Δ^+-web table** (denoted as W_{Δ^+}): contains a set of tuples containing the new nodes inserted into W_1 for transforming it into W_2. Note that this web table represents the **Insert Node** operation.
- **Δ^--web table** (denoted as W_{Δ^-}): contains a set of tuples containing nodes deleted from W_1 as determined by the **Delete Node** operation which transforms W_1 to W_2. Note that the web tuples in W_{Δ^-} do not necessarily indicate that these sets of inter-linked Web documents are deleted from the Web site. These documents may still exist in the Web, however they may no longer be relevant to the user's query due to modification of the content or inter-linked structure of these pages.
- **Δ^M-web table** (denoted as W_{Δ^M}): contains a set of web tuples that represents the previous and current sets of nodes modified by the **Update Node** operation.

Definition 1 [Delta Web Tables] *Let $W_1 = \langle Z_1, S_1, T_1 \rangle$ and $W_2 = \langle Z_2, S_2, T_2 \rangle$ be two web tables generated by a polling coupling query G at time t_1 and t_2. Let A, D and U be the sets of nodes added to, deleted from and updated during t_1 and t_2 to transform W_1 to W_2. Then,*

- *$W_{\Delta^+} = \langle Z_{\Delta^+}, S_{\Delta^+}, T_{\Delta^+} \rangle$ is called a* **Δ^+-web table** *where $S_{\Delta^+} = S_2$, $T_{\Delta^+} \subseteq T_2$ and for each web tuple $w_i \in T_{\Delta^+}$, $\forall\ 0 < i \leq |T_{\Delta^+}|$ there exists a node $n(w_i)$ such that $n(w_i) \notin T_1$ and $n(w_i) \in A$.*

- $W_{\Delta^-} = \langle Z_{\Delta^-}, S_{\Delta^-}, T_{\Delta^-} \rangle$ *is called a* **Δ^--web table** *where* $S_{\Delta^-} = S_1$, $T_{\Delta^-} \subseteq T_1$ *and for each web tuple* $w_i \in T_{\Delta^-}$, $\forall\ 0 < i \leq |T_{\Delta^-}|$ *there exists a node* $n(w_i)$ *such that* $n(w_i) \notin T_2$ *and* $n(w_i) \in D$.
- $W_{\Delta^M} = \langle Z_{\Delta^M}, S_{\Delta^M}, T_{\Delta^M} \rangle$ *is called a* **Δ^M-web table** *where* S_{Δ^M} *is generated from* S_1 *and* S_2 *and for each web tuple* $w_i \in T_{\Delta^M}$, $\forall\ 0 < i \leq |T_{\Delta^M}|$ *at least one of the following conditions must be true:*

  1. *If there exists a node* $n_1(w_i)$ *such that* $n_1(w_i) \in U$ *and* $n_1(w_i) \notin (D \cup A)$ *then there must exist a node* $n_2(w_i)$ *such that* `url(`$n_1(w_i)$`)` $=$ `url(`$n_2(w_i)$`)` *and* `date(`$n_1(w_i)$`)` $\neq$ `date(`$n_2(w_i)$`)`.
  2. *If there exists a node* $n_3(w_i)$ *such that* $n_3(w_i) \notin U$ *then* $n_3(w_i) \in (D \cup A)$ *must be true. In this case, there must not exist a node* $n_4(w_i)$ *such that* `url(`$n_3(w_i)$`)` $=$ `url(`$n_4(w_i)$`)`.
  3. *If there exists a node* $n_4(w_i)$ *such that* $n_4(w_i) \notin (D \cup A \cup U)$ *then there must not exist another node* $n_5(w_i)$ *such that* `url(`$n_4(w_i)$`)` $=$ `url(`$n_5(w_i)$`)`. ∎

Typically, the delta web tables reflect the *net effect* of Web site modification, that is, they contain only the net result of successive modification of a set of relevant documents in the Web. Note that in most of the cases the size of these delta web tables will be much smaller than W_1 or W_2. Representing changes in the form of a set of delta web tables enables us to view the history of a web table as a combination of a single web table snapshot and a collection of delta web tables. We can obtain various states of a web table by starting with a single web table and applying some sequence of web deltas to it. Also, to minimize the storage cost we materialize only a single web table and a set of delta web tables in lieu of the various states of the web table.

Observe that the representation of web deltas using delta web tables are comparable to delta relations in the relational model. In a relational database, deltas usually are represented using delta relations: For a relation R, delta relations *inserted(R)* and *deleted(R)* contain the tuples inserted to and deleted from R, while delta relations *old-updated(R)* and *new-updated(R)* contain the old and new values of updated tuples [26]. Similarly, we store the Web documents which are added, deleted or modified in Δ^+, Δ^- and Δ^M-web tables respectively.

4.4 Decomposition of Change Detection Problem

The problem of detecting and representing changes can now be decomposed into two parts:

1. Construction of the joined and outer joined web tables from W_1 and W_2. We discuss the construction of these web tables in the next section.

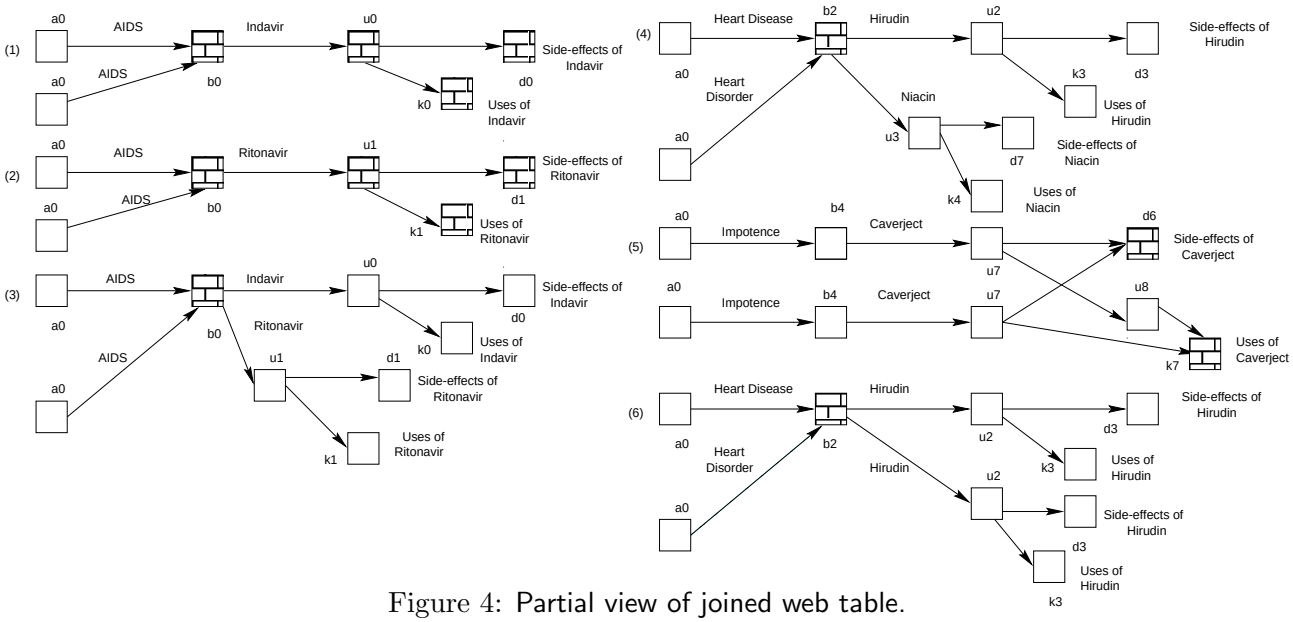


Figure 4: Partial view of joined web table.

2. Use the joined and outer joined web tables to generate a set of delta web tables, i.e., W_{Δ^+}, W_{Δ^-} and W_{Δ^M}, containing web deltas. We discuss this in Section 6.

5 Web Join and Outer Web Join

In this section, we briefly introduce the web join and outer web join operators. We first introduce these operators and then discuss their algorithm. Note that in this paper we discuss the web join and outer web join only to the extent it is necessary to understand the concept of change detection and representation in WHOWEDA. The reader may refer to [4] for details.

5.1 Web Join

The web join operator is used to combine two web tables by *joining* a web tuple of one table with a web tuple of other table whenever there exist joinable nodes. Let $w_a \in W_1$ and $w_b \in W_2$ be two web tuples. Then these tuples are joinable if there exist at least one node in w_a which is joinable to a node in w_b. The joined web tuple contains the nodes from both the input web tuples. We materialize the joined web tuple in a separate web table. As one of the joinable node in each joinable node pair is superfluous, we remove one of them from the joined web tuple.

To perform a web join operation on web tables W_1 and W_2, a pair of web tuples is selected, one from each web table, and all the pairs of nodes are evaluated to determine if there exist joinable nodes. The process is repeated for all $|W_1| \times |W_2|$ pairs of web tuples. If there exist joinable nodes in a pair of web tuples then the web tables are joinable. Formally, two web tables W_1 and W_2 are

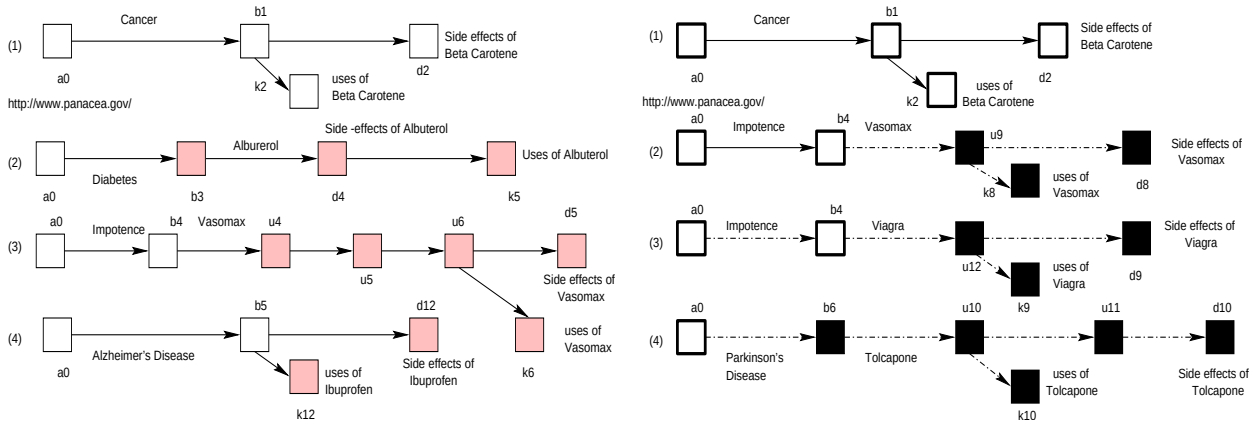
joinable if $w_a \in W_1$ and $w_b \in W_2$ are joinable where $0 < a \leq |W_1|$, $0 < b \leq |W_2|$. We express the web join between W_1 and W_2 as $W_{12} = W_1 \bowtie W_2$.

Example 1 Consider the web tables Drugs and New Drugs. The joined web table of these two web tables is shown in Figure 4. The nodes b_0, u_0, d_0 and k_0 in the first web tuple in Drugs are identical to those in the first web tuple in New Drugs as these nodes remain unchanged during the transition. The joined web tuple generated by concatenating these two web tuples over the nodes b_0, u_0, d_0 and k_0 is shown in Figure 4 (the first web tuple). Similarly, the second joined web tuple in Figure 4 is the result of joining the second web tuples in Drugs and New Drugs. The third joined web tuple is generated by joining the first web tuple in Drugs to the second web tuple in New Drugs over the node b_0, and so on. Observe that the third, fourth, fifth and sixth web tuples in Drugs do not participate in the web join process as the nodes in these web tuples are not identical to any nodes in New Drugs. ∎

5.2 Outer Web Join

The web tuples that do not participate in the web join operation (dangling web tuples) are absent from the joined web table. In certain situations it is necessary to identify the dangling web tuples from one or both of the input web tables. The outer web join operation enables us to identify them. Depending on whether the outer-joined web table must contain the non-participant web tuples from the first or second web tables, we define two kinds of outer web join: the *left-outer web join* and the *right-outer web join* respectively. Formally, given two web tables W_1 and W_2, the left-outer web join and right-outer web join on these two web tables are denoted by $W_1 =\bowtie W_2$ and $W_1 \bowtie= W_2$ respectively, where the symbols $=\bowtie$ and $\bowtie=$ corresponds to the different flavors of outer web join. The resultant web table W_o for a left-outer web join or right-outer web join will contain the dangling web tuples from W_1 or W_2 respectively.

Example 2 Consider the web tables Drugs and New Drugs in Figures 2 and 3 respectively. The web tuples in Drugs and New Drugs, which are associated with the side effects and uses of "Beta Carotene", a drug used for cancer (third web tuple), do not participate in the web join process as the content of all the nodes in the web tuple in New Drugs has changed with respect to those in Drugs. The link structure of the web tuple related to "Vasomax" has been modified after 15th January, 2001 and none of the nodes in this web tuple in Drugs are joinable to the corresponding web tuple in New Drugs. The web tuple related to "Alzheimer's Disease" in Drugs is not materialized again in New Drugs as the new set of documents does not satisfy the coupling query anymore. Similarly, the web tuple containing documents related to "Diabetes" in Drugs



(a) Left outer web join. (b) Right outer web join.

Figure 5: Outer web join.

Input: Set of node ids N_1 and N_2 of W_1 and W_2.
Output: Joinable node id set J.

(1) $U = N_1 \cap N_2$;
(2) for $(i = 1\ to\ |U|)$ {
(3) $v_1(i) =$ **GetVersionId**$(U(i), W_1)$; /* returns the version id of a node having id $U(i)$ in W_1 */
(4) $v_2(i) =$ **GetVersionId**$(U(i), W_2)$;
(5) if $(v_1(i) = v_2(i))$ /* The nodes are joinable */
(6) Insert $U(i)$ in joinable node id set J;
(7) else
(8) The nodes are not joinable;
(9) }
(10) return J;

Figure 6: Algorithm for computing joinable nodes.

has been removed from the Web site and is not materialized once again in New Drugs. These four web tuples in Drugs are dangling web tuples. Performing a left outer join on these two web tables enables us to identify these dangling web tuples (Figure 5(a)).

Now consider New Drugs. The last two web tuples dealing with "Viagra" and "Tolcapone" did not exist in the previous version as these drugs were added to the Web site after 15th January, 2001. Moreover, all the nodes in the web tuples related to "Vasomax" and "Beta Carotene" are modified. Thus, these tuples are dangling web tuples. Performing a right-outer web join on these two web tables enables us to identify these dangling web tuples in New Drugs (Figure 5(b)).

Observe that although the web tuple related to "Niacin" in New Drugs does not appear in Drugs, it is not a dangling web tuple as the node b_2 in this tuple is joinable with the corresponding node in the web tuple related to the drug "Hirudin" in Drugs. ∎

5.3 Algorithms of Web Join and Outer Web Join

In this section, we describe the algorithms for web join and outer web join operations. We first describe the algorithm for computing the joinable nodes in the web tables participating in web join (Figure 6). The algorithms of web join and outer web join operators use this algorithm to determine the joinable nodes.

The algorithm in Figure 6 takes as input the set of node ids in W_1 and W_2, denoted by N_1 and N_2, and returns as output the set of joinable node ids in W_1 and W_2 (denoted by J). Comparing the URLs and dates of each possible pair of Web pages represented by the nodes in N_1 and N_2 to identify the joinable nodes can have significant impact on the total cost and the query response time for large web tables to render web join impractical. A practical solution to this problem is to first subdivide the set of nodes in each web table into a disjoint and smaller subset U containing the ids of the nodes which occur in W_1 as well as in W_2. This is because a node can be potentially joinable only if it occurs in both the input web tables. Recall that two nodes have identical URLs if their ids are identical. Thus, each element in U is the identifier of a node that exists in W_1 and W_2. However, as nodes with identical URLs in different web tables may not always be identical in their contents, for each node id $u_k \in U$ the algorithm retrieves the version ids of u_k in W_1 and W_2 (denoted as $v_i(k)$ and $v_j(k)$ respectively). If the version ids are identical then the nodes represented by u_k in W_1 and W_2 are identical and are considered as joinable nodes. In that case, the algorithm inserts u_k in J. For example, consider the web tables Drugs and New Drugs. Suppose we wish to find the joinable nodes in these two web tables. The algorithm will output the set of joinable node ids in these two web tables, i.e., $J = \{b_0, b_2, u_0, u_1, d_0, d_6, d_1, k_0, k_1, k_7\}$.

The algorithms of web join and outer web join are given in Figures 7 and 8. Note that Step (20) in Figure 7 creates the resultant web tuple pool of the joined web table by concatenating the web tuples in $temp1$ and $temp2$ based on the joinable nodes J. Due to space constraints, we do not discuss the issues related to the construction of joined web tuples in detail. The reader may refer to [4] for details.

5.4 Complexity Analysis of Algorithms

In this section, we will analyze the time complexity of the algorithms in terms of the number of steps required for the algorithms in Figures 6, 7 and 8. Note that the analysis we present is the worst case analysis. We assume the following for the purpose of complexity analysis:

- The node-ids of a web table are stored in the web tuple pool in a B+-tree.
- The node-ids are also arranged in the node pool in a B+- tree. The version-ids are stored

Input: Web tables $W_1 = \langle Z_1, S_1, T_1 \rangle$ and $W_2 = \langle Z_2, S_2, T_2 \rangle$.
Output: Joined table W_{12}.

(1) $N_1 =$ **GetAllNodeIds**(W_1); /* returns a set of node ids in W_1 */
(2) $N_2 =$ **GetAllNodeIds**(W_2);
(3) $J =$ **ComputeJoinableNodeIds**(N_1, N_2);/* Figure 6 */
(4) for ($a = 1$ *to* $|W_1|$) { /* Select tuples containing joinable nodes */
(5) Get tuple w_a;
(6) $tupleNodeIdSet[a] =$ **GetTupleNodeIds**(w_a); /* returns a set of node ids in $w_a \in W_1$ */
(7) if ($tupleNodeIdSet[a] \cap J \neq \emptyset$)
(8) Store tuple w_a in temporary tuple pool $temp1$; /* The tuple is a joinable web tuple* /
(9) else
(10) w_a is not joinable;
(11) }
(12) for ($b = 1$ *to* $|W_2|$) {
(13) Get tuple w_b;
(14) $tupleNodeIdSet[b] =$ **GetTupleNodeIds**(w_b);
(15) if ($tupleNodeIdSet[b] \cap J \neq \emptyset$)
(16) Store tuple w_b in temporary tuple pool $temp2$;/* The tuple is a joinable web tuple */
(17) else
(18) w_b is not joinable;
(19) }
(20) $resultTuplePool =$ **JoinTuples**($J, temp1, temp2$); /* returns a set of joined web tuples by concatenating the tuples in $temp1$ and $temp2$ over the joinable nodes J */
(21) $resultantNodeIdSet =$ **GetAllNodeIds**($resultTuplePool$);
(22) for ($k = 1$ *to* $|resultantNodeIdSet|$) {
(23) Get node $resultantNodeIdSet[k]$ from table node pool of W_1 or W_2;
(24) Store the node in resultant table node pool of W_{12};
(25) }
(26) Create schema of W_{12} from S_1 and S_2;
(27) return W_{12};

Figure 7: Algorithm of web join.

with corresponding node-ids in the node pool. Note that the look-up time for each node-id is $O(log\ N)$ if we have N nodes in a web table stored as a B+- tree.

- Let n_1 be the number of nodes in web table W_1 and n_2 be the number of nodes in web table W_2. When $n_1 = n_2$, we say that both tables have the same number of nodes. That is, the number of nodes deleted from W_1 during transition is same as the number of nodes inserted in W_2.
- Let W_{t1} and W_{t2} be the average number of tuples in W_1 and W_2 respectively.
- Let N_{w1} and N_{w2} be the average number of nodes in each web tuple in W_1 and W_2 respectively.

1. Note that $n_1 = N_{w1} \times W_{t1}$ and $n_2 = N_{w2} \times W_{t2}$
2. The size of an I/O block is B = 8192 bytes.
3. The size of a tuple in the tuple pool is smaller than the I/O block so that each I/O access can obtain B/n_t tuples, where n_t is the average size of the tuple.

Input: Web tables $W_1 = \langle Z_1, S_1, T_1 \rangle$ and $W_2 = \langle Z_2, S_2, T_2 \rangle$.
Output: Left or right outer web joined table W_o.

(1) $N_1 =$ **GetAllNodeIds**(W_1);
(2) $N_2 =$ **GetAllNodeIds**(W_2);
(3) $J =$**ComputeJoinableNodeIds**(N_1, N_2); /* Figure 6 */
(4) if (left outer web join is to be performed) {
(5) $S_o = S_1$; /* Web schema of the result web table */
(6) $danglingNodeSet = N_1 - J$; /* Ids of nodes that do not participate in web join */
(7) Let temporary web table $W_t = W_1$;
(8) }
(9) else {
(10) $S_o = S_2$;
(11) $danglingNodeSet = N_2 - J$;
(12) Let temporary web table $W_t = W_2$;
(13) }
(14) for ($r = 1$ *to* $|W_t|$) {
(15) Get tuple w_r;
(16) $tupleNodeIdSet[r] =$ **GetTupleNodeIds**(w_r);
(17) if ($tupleNodeIdSet[r] \cap danglingNodeSet = tupleNodeIdSet[r]$)/* dangling web tuple */ {
(18) for ($d = 1$ *to* $|tupleNodeIdSet[r]|$) {
(19) Insert $tupleNodeIdSet[r][d]$ into $resultantNodeIdSet$;
(20) Store tuple w_r in web tuple pool of W_o;
(21) }
(22) }
(23) else
(24) w_r contains joinable nodes;
(25) }
(26) for ($k = 1$ *to* $|resultantNodeIdSet|$) {
(27) Retrieve the node having id $resultantNodeIdSet[k]$ from table node pool of W_t;
(28) Store the node in resultant table node pool of W_o;
(29) }
(30) return W_o;

Figure 8: Algorithm of outer web join.

4. The size of a node in the node pool is smaller than the size of the I/O block so one access can obtain B/n_s nodes, where n_s is the average size of the node in the node pool.

To get the node-ids from the web tuple pool, we need to access the tuples from the web tuple pool. Thus, the number of block accesses needed is $(W_{t1} \times N_{w1} \times n_t)/B$ and $(W_{t2} \times N_{w2} \times n_t)/B$ for W_1 and W_2 respectively. For accessing the nodes from the node pool , n_t will be replaced by n_s.

Analysis of the algorithm in Figure 6 to find joinable nodes: Step (1) of the algorithm will need $O(n_1 log\ n_2)$ steps, as each look up operation induces a time complexity of $O(log\ n_2)$. For simplicity, if we assume that the number of nodes in the two tables are same (that is, the number of nodes deleted is equal to the number of nodes which have been inserted during the transition

of Web page), then the complexity is $O(nlog\ n)$. For simplicity, let us assume that the number of joinable nodes be $n/2$ where n is the number of nodes in W_1 and W_2. That is, half of the nodes remain unchanged during the transition. Hence, Step (3) will be performed $n/2$ times. To get the version-ids associated with each node-id of each node from the node pool will take $log\ N$ steps where N is the total number of nodes in the node pool (i.e., total nodes in the warehouse). Thus, Steps (4) and (5) will induce complexity of $n/2 \times (O(log\ N) + O(log\ N))$, which is $O(nlog\ N)$. Hence, the total complexity is $O(nlog\ N)+O(nlog\ n)$, which is $O(n(log\ N+log\ n))$. Since $n < N$, the complexity will be $O(nlog\ N)$. Note that the I/O cost involve here is negligible as only ids have been accessed to determine the joinable nodes.

Analysis of the web join algorithm in Figure 7: For simplicity, here again we assume that both the tables have n nodes. Steps (1) and (2) both together will induce time complexity of $O(2log\ n)$. Thus, the total complexity of the first three steps to find joinable nodes is $(O(nlog\ N)+ O(nlog\ n)+O(2log\ n))$, which is $(O(nlog\ N)+O((n+2)log\ n))$. Since $n < N$, it will be $O(nlog\ N)$. Step (4) will be executed W_{t1} times. Step (5) will need $(N_{w1} \times n_t)/B$ I/O accesses for each tuple. Steps (6) and (7) will need $O(nlog\ (N_{w1})/2)$ steps where N_{w1} is the average number of nodes in a web tuple in W_1 and we assumed that there are $n/2$ joinable nodes. Thus, the time complexity of Steps (4) to (7) will be $W_{t1} \times (O(nlog\ (N_{w1})/2))$ plus the I/O cost of accessing $(W_{t1} \times N_{w1} \times n_t)/B$ blocks of memory for all the web tuples of W_1. Similarly, there will be $W_{t2} \times (O(nlog\ (N_{w2})/2))$ steps to find the joinable tuples from W_2 plus the I/O cost. Step (20) will take $J_N \times (n/2)$ steps where J_N is the number of joinable tuples and $n/2$ is the average number of nodes in the two web tables. Here, we assume that indexes are built in the storage structure on the joinable nodes so that the cost of accessing them is $O(1)$. The cost of Step (21) can be calculated as in Step (1) or (2). Step (22) will take $J_N \times N_{JN}$ where N_{JN} is the average number of nodes in each joinable tuples. Step (23) will take $log\ (n_1 + n_2)/2$ steps in the worst case of searching a node in both the table pools plus n_s/B block accesses for fetching each node. Step (24) will take $log\ (J_N \times N_{JN})$ on the average to insert a node in the node pool tree plus n_s/B I/O accesses for each node. Note that we do not calculate the time complexity of Step (26) as the schema generation process does not influence the web delta detection problem and hence it is beyond the scope of the paper. For change detection problem, Step (26) may be ignored. The reader may refer to [6] for discussion on the schema generation process. Thus, to sum up, the web join process induces both the operational complexity and the I/O access cost. The accumulated operational complexity after assuming that $n_1 = n_2$, $(W_{t1} = W_{t2}) = W_t$, $(N_{w1} = N_{w2}) = N_w$, is $(O(nlog\ N)+2 \times W_t \times (O(nlog\ (N_w)/2))+J_N \times (n/2)+(J_N \times N_{JN} \times log(J_N \times N_{JN})(1+O(log\ n))$.

By assuming the values of W_t, N_w, J_N, N_{JN} to be very small as compared to n and since n is smaller than N, we get the operational complexity as $C \times O(nlog\ N)$, where C is a constant. The total I/O cost is $((2/B) \times ((n \times n_t) + (J_N \times N_{JN}) \times n_s))$.

Analysis of the outer web join algorithm: The number of steps needed in the algorithm in Figure 8 to find outer web join will be the same as that for finding the joinable web tuples. This is because the number of dangling nodes (nodes which are not joinable) will be $n/2$ as we assumed that the joinable node set is of size $n/2$. Thus, to find the web tuples in each left outer and right out web join will need the same number of steps as in the case of the joinable web tuples described above.

The complexity of all other algorithms in the next section can be calculated on similar lines since they all use similar type of constructs and reasoning. Therefore, they are left for the readers to verify.

6 Generating Delta Web Tables

This section initiates a discussion on the algorithm to detect and represent different types of change operations using the web join and outer web join operations. We describe the Algorithm *Delta* that generates a set of delta web tables to detect and represent web deltas. We first describe the generation of delta tables informally and then provide the complete algorithm.

6.1 Outline of the Algorithm

The algorithm for delta web table generation can be best described by the following four phases: the *join tables generation phase*, the *delta node identification phase*, the *delta tuples identification phase* and the *delta table generation phase*. We discuss these phases one by one.

Phase 1: Join Tables Generation Phase: This phase takes as input two web tables, new and old versions, and generates the joined, right outer joined and left outer joined web tables. For instance, after this phase the web tables in Figures 4, 5(a) and 5(b) are generated from Drugs and New Drugs.

The right outer join operation on W_1 and W_2 may create three categories of dangling web tuples: (1) Web tuples which are added to W_1 during the polling times t_1 and t_2. These tuples may contain some new nodes and the contents of the remaining ones have been changed. (2) Tuples in which all the nodes have undergone content modification. (3) Tuples in which some of the nodes are new and the contents of the remaining ones have changed but these tuples existed in W_1. For example, consider the web table in Figure 5(b). The last two web tuples belong to

the first category. The first web tuple belongs to the second category and the second web tuple is an example of third category.

The web join operation on W_1 and W_2 may contain the following three types of web tuples: (1) Web tuples in which all the nodes are joinable nodes. These tuples are the results of joining two versions of web tuples in W_1 and W_2 that have remained unchanged during t_1 and t_2. (2) Web tuples in which some of the nodes are joinable nodes and remaining nodes are the result of insertion, deletion or modification operations during the transition. (3) Some of the nodes are joinable nodes and out of the remaining ones, some are the result of insertion, deletion or modification and the remaining ones are not joinable in this web tuple but they have remained unchanged during the transition. That is, these nodes may be joinable nodes in some other joined web tuple(s). While generating delta web tables, the algorithm ignores the first category of web tuples in the joined web table as it does not reflect any changes. For instance, in the joined web table in Figure 4, all the web tuples are of second and third categories. Specifically, the first two and the last three tuples contain nodes whose contents are modified. The fourth web tuple contains nodes whose contents are modified as well as a node k_4 which is inserted during t_1 and t_2. Finally, the fifth tuple contains a node u_8 which is deleted as well as a set of nodes which are modified. Hence, these web tuples represent the second category. On the other hand, the third web tuple represents the third category. This is because a_0 represents a modified node and the nodes u_0, u_1, d_0, d_1, k_0 and k_1 are not joinable in this web tuple but they are joinable nodes in the first and second web tuples.

Similar to the right outer joined web table, the left outer joined table may contain the following three categories of web tuples: (1) Web tuples which are deleted from W_1. These tuples do not occur in W_2. (2) Tuples in which every node has undergone content modification. (3) Tuples in which some nodes are deleted from W_1 and remaining ones have been modified. The new and old versions occur in both the tables in W_1 and W_2. For instance, the second and fourth web tuples in Figure 5(a) belong to the first category. The first and the third web tuples belong to the second and third categories respectively.

Phase 2: Delta Nodes Identification Phase: In this phase, the nodes which are added, deleted or modified during t_1 and t_2 are identified. This phase takes as input the web tables W_1 and W_2 and the set of joinable nodes from the joined table and generates sets of nodes which are added, deleted or modified during the time interval. Thus, nodes which exist in W_2 but not in W_1 are the new nodes that are added to W_1. Similarly, nodes which only exist in W_1, but not in W_2 are the nodes that are removed from W_1. Furthermore, the nodes which are not joinable

nodes, but they exist in W_1 as well as W_2 are essentially the nodes that have undergone content modification during t_1 and t_2. For instance, $\{b_3, u_4, u_5, u_6, u_8, d_4, d_5, d_{12}, k_5, k_6, k_{12}\}$ are the ids of nodes which appears in Drugs but not in New Drugs. Hence, these nodes were removed during transition. Similarly, $\{k_4, u_9, k_8, d_8, d_7, u_3, u_{12}, d_9, k_9, u_{10}, u_{11}, d_{10}, k_{10}\}$ are the ids of nodes that exist in New Drugs but not in Drugs. Hence, these nodes were added during t_1 and t_2. Finally, the nodes with ids a_0, b_1, d_2, k_2, u_2, d_3, k_3, b_4 and u_7 appear in both the web tables but are not joinable. Hence, these nodes have undergone content modification.

Observe that at this point we have identified the nodes which are inserted, deleted or modified during the transformation. Next, the algorithm proceeds to determine how these nodes are related to one another and how they are associated with those nodes which have remained unchanged during t_1 and t_2.

Phase 3: Delta Tuples Identification Phase: In the delta tuples identification phase we are interested in identifying those web tuples which contain nodes which are added, deleted or modified during the transition. It should be clear that we are not just simply identifying these web tuples as it can be done by inspecting W_1 and W_2 without performing any web join or outer web join operations. Our objectives are the following:

- In case of added or deleted nodes, we are not simply interested in identifying tuples containing these nodes only but also how these nodes are linked to or related to the existing nodes (nodes which prevailed during the transition). Moreover, we wish to determine how the new or deleted nodes are related to one another.
- For nodes which have undergone content modification during the transition, we wish to determine how these nodes are linked to one another and to those nodes which have remained unchanged. We also wish to present the old and new versions of the nodes in a single tuple so that a user can view them effectively. Finally, we wish to highlight the changes in the overall hyperlink structure due to the content modification.

To achieve this, we scan the joined and outer joined web tables. The delta tuples identification phase takes as input these web tables and the sets of nodes which are added, deleted or modified. It returns as output sets of tuples containing nodes which are added, deleted and modified respectively. In the remaining portion of this paper, these sets are denoted as $insertTupleSet$, $deleteTupleSet$ and $updateTupleSet$ respectively. Observe that the nodes which are added during the transition can occur in the following tables:

- In the right outer joined table if the remaining nodes in a tuple containing the new nodes are

modified and hence, are not joinable. The second, third and fourth web tuples in Figure 5(b) are examples of such tuples.

- In the joined web table if some of the nodes in the tuple containing new nodes have remained unchanged during the transition and hence are joinable. For instance, the fourth web tuple in Figure 4 is an example of such a web tuple where k_4, u_3 and d_7 are the new nodes and b_2 has remained unchanged during the transition and therefore joinable.

Hence, we need to scan the joined and right outer joined tables to identify the tuples containing nodes which are inserted during t_1 and t_2. Similarly, the nodes which are deleted during the transition may occur in the following two web tables:

- In the left outer joined table if the remaining nodes in a tuple containing the deleted nodes are modified and hence, are not joinable. The second, third and fourth web tuples in Figure 5(a) are examples of such tuples.
- In the joined web table if some of the nodes in the tuple containing deleted nodes have remained unchanged during the transition and hence are joinable. For instance, the seventh web tuple in Figure 2 contains the node u_8 which is deleted. As the nodes d_6 and k_7 have remained unchanged during the transition, these nodes are joinable. The tuple containing this deleted node can be detected from the joined web table in Figure 4 (third web tuple).

Thus, the algorithm scans the left outer joined and joined tables to retrieve the tuples containing deleted nodes. Finally, the nodes which are modified during the transition can be identified by inspecting all the three web tables:

- Tuples in the left and right outer joined tables which do not contain any new or deleted node represent the old and new versions of these nodes respectively. These web tuples do not occur in the joined table as all the nodes are modified. For instance, the first web tuples in Figures 5(a) and (b) are examples of such tuples.
- Tuples in the left and right outer joined tables that contain modified nodes as well as inserted or deleted nodes. Note that these modified nodes may not appear in the joined web table if no other joinable web tuples contain these modified nodes.
- Tuples in the joined web tables whose some of the nodes represent the old and new versions of these modified nodes. For instance, the first web tuple in Figure 4 contains the old and new versions of a_0.

Phase 4: Delta Web Tables Generation Phase: Finally, the three types of delta web tables are generated in this phase. It takes as input the three sets of tuples, i.e., $insertTupleSet$,

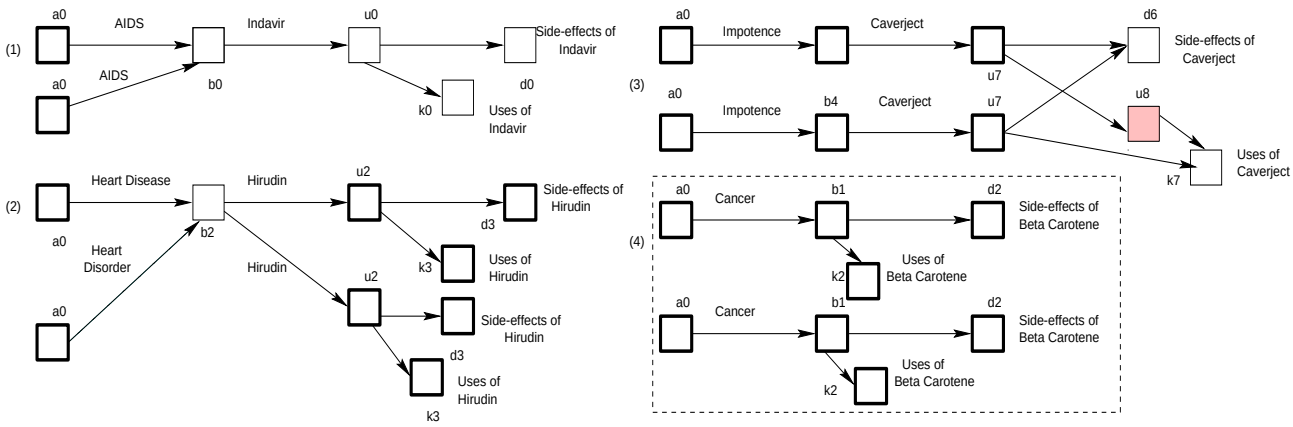


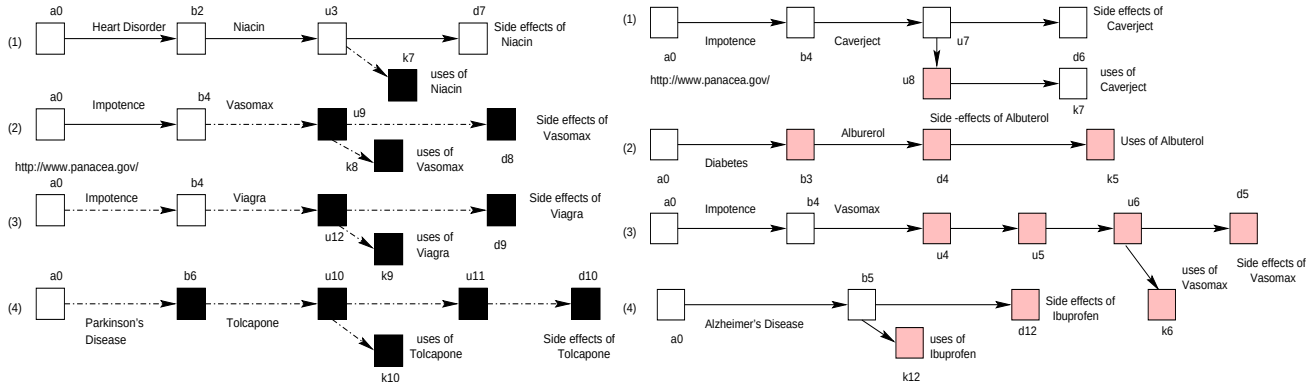
Figure 9: Δ^M-web table.

deleteTupleSet and *updateTupleSet*, generated in the previous phase and generates the delta web tables from these sets. The procedure to generate these tables is straightforward. The tuples in *insertTupleSet* are stored in Δ^+-web table. The tuples in *deleteTupleSet* and *updateTupleSet* are stored in Δ^- and Δ^M-web tables respectively.

Next, we illustrate the generation of delta web tables informally with an example given below.

Example 3 Consider the two web tables Drugs and New Drugs in Figures 2 and 3. We would like to find the various change operations that transform Drugs into New Drugs. Changes may include, inserting nodes, deleting nodes and updating nodes in Drugs. For each type of these changes, we create the W_{Δ^+}, W_{Δ^-} and W_{Δ^M} tables. We discuss the generation of each delta web tables in turn. Figure 9 depicts the Δ^M-web table. The patterned boxes in this figure in each web tuple are the old and new versions of the nodes. For example, the second web tuple in Figure 9 contains the old and new versions of the nodes a_0, u_2, d_3 and k_3, along with the joinable node u_2 (content of u_2 has remained unchanged during the transition). Each web tuple shows how the set of modified nodes is related to one another and with the joinable nodes. Observe that the first four web tuples are extracted from the joined web table in Figure 4. The last web tuple (enclosed in a dotted box) is the result of the integration of two web tuples - one from the left outer joined web table in Figure 5(a) and another from the right outer joined table in Figure 5(b).

Figure 10(a) illustrates the Δ^+-web table. The black boxes in each web tuple are the new nodes inserted into Drugs during 15th January, 2001 and 15th February, 2001. Similar to Δ^M-web table, each web tuple in Δ^+-web table shows how the new nodes are related to other relevant nodes in the web table. Note that the last three web tuples in Figure 10(a) are extracted from the right outer joined web table in Figure 5(b). However, as the node b_2 in the first web tuple is a joinable node, the new nodes k_4, u_3 and d_7 in this tuple are identified from the fourth web



(a) Δ^+-web table. (b) Δ^--web table.

Figure 10: Δ^+ and Δ^- web tables.

tuple of the joined web table in Figure 4.

Finally, Figure 10(b) depicts the Δ^--web table containing all the nodes that are deleted from Drugs. The last three web tuples are extracted from the web table in Figure 5(a). However, the tuple containing the deleted node u_8 is extracted from the fifth web tuple in the joined web table in Figure 4. Observe that we do not materialize the joined web tuples containing new or deleted nodes in W_{Δ^+} and W_{Δ^-} respectively. Instead, we extract the original web tuple containing these nodes from the joined web tuple and materialize them in W_{Δ^+} and W_{Δ^-} respectively. ∎

6.2 Algorithm Delta

We now provide the formal algorithm for the four phases discussed in the previous section. We describe how, given two web tables W_1 and W_2 created by a polling global coupling operation at polling times t_1 and t_2, we compute a set of delta web tables corresponding to various types of changes to transform W_1 to W_2. The pseudocode for this algorithm is given in Figure 11. It takes as input two web tables $W_1 = \langle Z_1, S_1, T_1 \rangle$ and $W_2 = \langle Z_2, S_2, T_2 \rangle$ created by a coupling query Q at polling times t_1 and t_2 respectively. It returns as output a set of delta web tables $W_{\Delta^+} = \langle Z_{\Delta^+}, S_{\Delta^+}, T_{\Delta^+} \rangle$, $W_{\Delta^-} = \langle Z_{\Delta^-}, S_{\Delta^-}, T_{\Delta^-} \rangle$ and $W_{\Delta^M} = \langle Z_{\Delta^M}, S_{\Delta^M}, T_{\Delta^M} \rangle$. The steps to generate these delta web tables are as follows.

6.2.1 Algorithm for Phases 1 and 2

Steps (1) to (4) implement the first phase of the algorithm and generates the joined, the right and the left outer joined web tables from W_1 and W_2 (Line (4)). In the algorithm, these web tables are denoted as W_j, W_{ro} and $W_{\ell o}$ respectively. Steps (5) to (7) implement the second phase of the

Input: Web tables $W_1 = \langle Z_1, S_1, T_1 \rangle$, $W_2 = \langle Z_2, S_2, T_2 \rangle$.
Output: Delta web tables $W_{\Delta^+} = \langle Z_{\Delta^+}, S_{\Delta^+}, T_{\Delta^+} \rangle$, $W_{\Delta^-} = \langle Z_{\Delta^-}, S_{\Delta^-}, T_{\Delta^-} \rangle$ and $W_{\Delta^M} = \langle Z_{\Delta^M}, S_{\Delta^M}, T_{\Delta^M} \rangle$.

(1) $N_1 =$ **GetAllNodeIds**(W_1); /* Phase 1 */
(2) $N_2 =$ **GetAllNodeIds**(W_2);
(3) $J =$ **ComputeJoinableNodeIds**(N_1, N_2); /* Figure 6 */
(4) $W_{12} =$ **GenerateResultTables**(N_1, N_2, W_1, W_2, J) where $W_{12} = \{W_{ro}, W_{lo}, W_j\}$;
(5) $delNodeSet = N_1 - N_2$; /* Ids of the nodes deleted from W_1 */ /* Phase 2 */
(6) $addNodeSet = N_2 - N_1$; /* Ids of the nodes added to W_1 */
(7) $updateNodeSet = (N_1 - J - delNodeSet) \cup (N_2 - J - addNodeSet)$;
(8) $N_j =$ **GetAllNodeIds**(W_j); /* Phase 3 */
(9) Let $K = updateNodeSet - (N_j - J)$;
(10) Let $A = addNodeSet$;
(11) Let $D = delNodeSet$;
(12) Let $U = (N_j - J) \cap updateNodeSet$;
(13) $insertTupleSet =$ **DeltasFromRightOuter**$(A, K, W_{ro}, temp1, temp2)$; /* Figure 12*/
(14) $deleteTupleSet =$ **DeltasFromLeftOuter**$(D, K, W_{lo}, temp1, temp2)$;
(15) if ($|A| = 0$ and $|D| = 0$) { /* checks if all the inserted and deleted nodes are identified */
(16) $updateTupleSet =$ **DeltasFromJoin**(W_j, U, N_1, N_2); /* Figure 13 */
(17) else
(18) $updateTupleSet =$ **DeltasFromJoin**$(A, D, U, W_j, insertTupleSet, deleteTupleSet)$; /* Figure 14 */
(19) if ($|temp1| \neq 0$)
(20) Insert into $updateTupleSet$ web tuples from $temp1$;
(21) if ($|temp2| \neq 0$)
(22) Insert into $updateTupleSet$ web tuples from $temp2$;
/* Phase 4 */
(23) $W_{\Delta^M} =$ **CreateDeltaM**$(updateTupleSet, W_1, W_2, S_{12}, updateNodeSet)$;
(24) $W_{\Delta^+} =$ **CreateDeltaPlus**$(insertTupleSet, W_1, W_2, addNodeSet)$;
(25) $W_{\Delta^-} =$ **CreateDeltaMinus**$(deleteTupleSet, W_1, W_2, delNodeSet)$;
(26) return W_{Δ^+}, W_{Δ^-} and W_{Δ^M};

Figure 11: Algorithm Delta.

algorithm and identify the node ids which are added to W_1, node ids which are removed from W_1 and ids of the nodes that have undergone content modification during t_1 and t_2.

6.2.2 Algorithm of Phase 3

We now discuss the algorithm for implementing the delta web tuples identification phase. To determine the association of nodes (represented by the identifiers in $addNodeSet$, $delNodeSet$ and $updateNodeSet$) with each other and with other relevant nodes in W_1, we identify the web tuples in W_j, W_{ro} and $W_{\ell o}$ containing these nodes and store them in the sets of web tuples denoted in the algorithm as $insertTupleSet$, $deleteTupleSet$ and $updateTupleSet$ respectively. Each element in $insertTupleSet$ and $deleteTupleSet$ are web tuples containing nodes that are inserted to or deleted from W_1 during t_1 and t_2. Each element in $updateTupleSet$ is an *integrated* web tuple containing old and new versions of the nodes which have undergone content modification. Note

that these sets of web tuples encapsulate the various change operations introduced in Section 4.

Steps (8) to (22) in Figure 11 present the pseudocode for the delta web tuples identification phase. Step (9) computes those nodes which are updated during the transition but are not captured by the joined web table. The set K contains ids of those updated nodes which do not exist in N_j; i.e., K represents those updated nodes which are not present in W_j but in W_{ro} and $W_{\ell o}$. If n is a node in K, then n does not occur in W_j. That is, the tuple containing n is a dangling web tuple and consequently is ignored by the web join operation. That is, if n represents a node which has undergone update during t_1 and t_2 then the web tuples containing n in W_1 and W_2 must be dangling web tuples. Otherwise, n must occur in the joined web table. As a result, the tuples containing n in W_1 and W_2 must be captured by the left and right outer joined web tables respectively (W_{ro} and $W_{\ell o}$). If $K = \emptyset$, then all updated nodes are captured by the joined web table.

Next, (Steps (10) and (11)) the sets of node ids of new and deleted nodes are copied to the sets A and D respectively. This is because subsequently A and D need to be updated every time we identify new and deleted nodes while scanning the right and outer joined web tables. However, we do not wish to modify the *addNodeSet*, *delNodeSet* and *updateNodeSet* since the algorithm is going to use them again to generate the delta web tables in Steps (23) to (25). Finally, U (Step (12)) represents those modified nodes which occur only in the joined web table W_j, i.e., $U \cap K = \emptyset$. Then, Steps (13) and (14) scan the right and left outer joined web tables to identify the web tuples containing inserted, deleted and updated (if any) nodes. We elaborate on these algorithms now.

Algorithm of DeltasFromRightOuter(A, K, W_{ro}, $temp1$, $temp2$) **(Step (13) in Figure 11)**

This algorithm takes as input the set of node ids which are added during the transition, i.e., A, the set K, the right outer joined web table W_{ro}, and two empty sets $temp1$ and $temp2$ to store *specific* web tuples. It returns as output a set of tuples containing the nodes which are inserted during t_1 and t_2, denoted by *insertTupleSet*, modified A and K, and $temp1$ and $temp2$ (possibly non-empty). Let us elaborate on the purpose of the tuple sets $temp1$ and $temp2$. $temp1$ stores those web tuples in the right or left outer joined tables that do not contain any new or deleted nodes (Category 2 web tuples as discussed in the previous section). That is, the tuples in $temp1$ represent those web tuples in which all the nodes content are modified during the transition. For instance, $temp1$ will store the first web tuples in Figures 5(a) and (b). Recall that in *updateTupleSet*, the old and new versions of a web tuple are integrated and stored together. However, the right outer

```
Input: Right outer-joined table W_ro, set of added and updated
node ids A and U respectively, temporary sets temp1 and temp2 to store tuples.
Output: Set of web tuples insertTupleSet containing new nodes.

(1)   for (b = 1 to |W_ro|) { /* Case 1: Detecting web deltas from W_ro */
(2)       Get tuple w_b;
(3)       tupleNodeIdSet[b] = GetTupleNodeIds(w_b);
(4)       if (tupleNodeIdSet[b] ∩ A ≠ ∅) {
              /* w_b contains node(s) which were inserted during the transition */
(5)           Store w_b in insertTupleSet;
(6)           Insert (tupleNodeIdSet[b] ∩ A) in tempAddSet; /* represents new nodes in w_b */
(7)           if (K ≠ ∅) {
              /* identify web tuples which contain modified nodes which are not present in the joined table */
(8)               if (tupleNodeIdSet[b] ∩ K ≠ ∅)
(9)                   Insert w_b in temp2;
(10)          }
(11)      }
(12)      else
(13)          Store w_b in temp1; /* tupleNodeIdSet[b] ∩ A = ∅ */
(14)  }
(15)  A = A − tempAddSet; /* New nodes which are already identified are now removed from A */
(16)  return temp1, temp2, insertTupleSet, A, K;
```

Figure 12: Algorithm for **DeltasFromRightOuter**(A, K, W_{ro}, $temp1$, $temp2$).

join operation only identifies the new version of the modified web tuple. The old version of the tuple can be extracted from the result of the left outer join operation. Thus, the insertion of the web tuples containing modified nodes into $updateTupleSet$ is deferred to the execution of the inspection of the left outer joined web table. Consequently, we store these tuples temporarily in $temp1$.

On the other hand, $temp2$ contains those tuples from the left or right outer joined tables which contain the updated nodes not captured by the joined web table. That is, $temp2$ contains tuples from W_{ro} and W_{lo} where each tuple contains at least one node that is an element of K and some of the remaining nodes must be new or deleted nodes. Notice the differences between the tuples in $temp1$ and $temp2$. In $temp1$, all the nodes in each web tuple have undergone content modification. However, in $temp2$ each web tuple must contain at least one node which is added or deleted during the transition and remaining nodes must have undergone content modification. Also observe that $temp1 \cap temp2 = \emptyset$. Note that the nodes in a tuple in $temp1$ may also occur in K, However, it is not captured in $temp2$ because each web tuple in $temp2$ contains one or more new or deleted nodes. Specifically, $temp2$ enables us to capture the web tuples containing modified nodes that cannot be identified from the joined web tables. For similar reasons as explained in the case of $temp1$, we defer the insertion of these tuples in the $updateTupleSet$. The pseudocode of this algorithm is given in Figure 12.

```
Input:Joined table W_j, set of updated node ids U, N_1 and N_2.
Output: Modified updateTupleSet.

(1)     for (a = 1 to |W_j|) {
(2)         Get tuple w_a;
(3)         tupleNodeIdSet[a] = GetTupleNodeIds(w_a);
(4)         Let X = tupleNodeIdSet[a] ∩ U;
            /* Identify the relevant tuples in the input web tables which contain the updated nodes */
(5)         Retrieve node set N_1(w_a) such that N_1(w_a) ⊆ X and N_1(w_a) ⊂ N_1;
(6)         Retrieve node set N_2(w_a) such that N_2(w_a) ⊆ X and N_2(w_a) ⊂ N_2;
(7)         if (X − (N_1(w_a) ∩ N_2(w_a)) = ∅) {
            /* tuple contains only both the old and new version of the updated nodes */
(8)             Store w_a in updateTupleSet;
(9)             /* Category 2 type joined web tuple */
(10)        }
(11)        else
(12)            w_a is ignored;
(13)    }
(14) }
(15) Return updateTupleSet;
```

Figure 13: Algorithm of **DeltasFromJoin**(W_j, U, N_1, N_2).

Algorithm of DeltasFromLeftOuter(D, K, W_{lo}, $temp1$, $temp2$) **(Step (14) in Figure 11)**

Next, the algorithm Delta inspects the left outer joined table $W_{\ell o}$ to identify the deleted or modified nodes. The pseudocode for this algorithm will be similar to Figure 12 and hence, it is omitted.

Algorithm of DeltasFromJoin(W_j, U, N_1, N_2) **(Step (16) in Figure 11)**

Finally, the Algorithm Delta in Figure 11 proceeds to inspect the joined web table W_j. If $A = D = \emptyset$, then the joined web table will only contain the updated nodes. Consequently, Step (16) is executed. Each joined web tuple in W_j is inspected to determine the existence of the old and new versions of the modified nodes. It may seem that each tuple containing dangling node(s)(nodes which are not joinable) may represent the old and new versions of the modified nodes. However, such assumption is not true. Note that we are specifically interested in those joined web tuples which contain only both the old and new versions of the updated nodes. Note that not all joined web tuples may satisfy this condition. For example, consider the joined web table in Figure 4. The fourth web tuple contains the dangling nodes u_2, k_3, d_3 etc.. However, both the old and new versions of these nodes are missing in this tuple. Specifically, it exists in the last joined web tuple. Hence, the last web tuple is inserted in *updateTupleSet* but not the fourth web tuple. Observe that this condition is checked in Step (7) of the algorithm in Figure 13. For instance, for the last web tuple, $N_1(w_a) = \{a_0, u_2, d_3, k_3\}$, $N_2(w_a) = \{a_0, u_2, d_3, k_3\}$ and $X = \{a_0, u_2, d_3, k_3\}$. Hence,

Input: Joined table W_j, set of added, deleted and updated node ids A, D and U respectively, $deleteTupleSet$, $insertTupleSet$.
Output: Modified $deletedTupleSet$, $insertTupleSet$, $updateTupleSet$.

```
(1)   for (a = 1 to |W_j|) {
(2)       Get tuple w_a;
(3)       tupleNodeIdSet[a] = GetTupleNodeIds(w_a);
(4)       Let Y = tupleNodeIdSet[a] − (tupleNodeIdSet[a] ∩ J);
(5)       if ((Y ≠ ∅) or (Y ∩ A ≠ ∅) or (Y ∩ D ≠ ∅)){
(6)           Retrieve node set N_1(w_a) such that N_1(w_a) ⊆ Y and N_1(w_a) ⊂ N_1;
(7)           Retrieve node set N_2(w_a) such that N_2(w_a) ⊆ Y and N_2(w_a) ⊂ N_2;
(8)           if (Y − (N_1(w_a) ∩ N_2(w_a)) = ∅) {
(9)               Store w_a in updateTupleSet;
(10)          }
(11)          else {
(12)              if ((Y − (N_1(w_a) ∩ N_2(w_a))) ⊂ A ∪ D) {
                      /* checks if the remaining dangling nodes are actually new or deleted nodes. */
(13)                  Extract w_2 ∈ W_2 from w_a; /* Original web tuple in W_2 which is in w_a is extracted */
(14)                  Extract w_1 ∈ W_1 from w_a;
(15)                  Insert w_a in updateTupleSet;
(16)                  Insert w_1 or w_2 to deleteTupleSet or insertTupleSet;
(17)              }
(18)              else {
(19)                  if (A ∩ [Y − (N_1(w_a) ∩ N_2(w_a))] ≠ ∅)
(20)                      Insert w_2 in insertTupleSet;
(21)                  if (D ∩ [Y − (N_1(w_a) ∩ N_2(w_a))] ≠ ∅)
(22)                      Insert w_1 in deleteTupleSet;
(23)              }
(24)              A = A − (A ∩ [Y − (N_1(w_a) ∩ N_2(w_a))]);
(25)              D = D − (D ∩ [D − (N_1(w_a) ∩ N_2(w_a))]);
(26)          }
(27)      }
(28)      else
(29)          w_a is ignored;
(30)  }
(31)  Return insertTupleSet, deleteTupleSet, updateTupleSet;
```

Figure 14: Algorithm of **DeltasFromJoin**(A, D, U, W_j, $insertTupleSet$, $deleteTupleSet$).

$\{X - (N_1(w_a) \cap N_2(w_a))\} = \emptyset$ and the joined tuple is inserted in $updateTupleSet$. However, for the fourth web tuple, $N_1(w_a) = \{a_0, u_3, k_3, d_3\}$, $N_2(w_a) = \{a_0, u_3, d_7, k_4\}$ and $X = \{a_0, u_2, d_3, k_3\}$. Hence, $\{X - (N_1(w_a) \cap N_2(w_a))\} = \{u_2, k_3, d_3\}$. Consequently, the condition is not satisfied.

Algorithm of DeltasFromJoin(A, D, U, W_j, $insertTupleSet$, $deleteTupleSet$) **(Step (18) in Figure 11)**

If all the new or deleted nodes are not identified after scanning W_{ro} and $W_{\ell o}$ then $|A| \neq 0$ or $|D| \neq 0$. Hence, in that case Step (18) in Figure 11 is executed. The pseudocode for the construct **DeltasFromJoin** is given in Figure 14. Note that in this case the algorithm is not only looking for the web tuples containing the updated nodes but also those web tuples which contain the new

or deleted nodes. We elaborate on these steps now. Note that Steps (2) to (10) are similar to the previous steps. Steps (12) to (25) are executed if not all the dangling nodes in w_a represent the new and old versions of the node. We explain these steps with examples. Consider the fifth web tuple in Figure 4. Here $Y = \{a_0, b_4, u_7, u_8\}$, $N_1(w_5) = N_2(w_5) = \{a_0, b_4, u_7\}$. Hence, $Y - (N_1(w_5) \cap N_2(w_5)) = \{u_8\}$. Hence, the condition in Step (8) is evaluated false. As u_8 represents deleted node, i.e., $u_8 \in D$. Consequently, the condition in Step (12) is evaluated true. We insert the tuple in *updateTupleSet* and extract the seventh web tuple in Drugs. Observe that the seventh web tuple in Drugs contains u_8 which is deleted during transition. Finally, we insert this tuple in the *deleteTupleSet*.

At this point the web tuple containing the new nodes k_4, u_3 and d_7 in the joined web table (fourth joined tuple) have not been identified yet. Note that for this web tuple $Y = D - (N_1(w_4) \cap N_2(w_4)) = \{u_2, k_3, d_3, k_4, u_3, d_7\}$. Also, $Y \subset A$ or $Y \subset D$ is not satisfied. Consequently, the condition in Step (12) is evaluated false. In order to identify this node, Steps (18) to (23) are executed. The condition in Step (19) is satisfied by Y as $Y \cap A = \{k_4, u_3, d_7\}$. Hence, Step (20) is executed and the original web tuple (fifth web tuple in New Drugs) is retrieved and inserted in *insertTupleSet*. If $Y \cap D \neq \emptyset$, then Step (22) is executed and the original web tuple from W_1 is retrieved and inserted in *deleteTupleSet*. Steps (24) and (25) update A and D by removing those ids which are already used to identify the web tuples.

We have now identified all the web tuples containing the new or deleted nodes. We have also identified some of the tuples containing the updated nodes. The remaining tuples containing the updated nodes are identified from *temp*1 and *temp*2 (Steps (19) to (22) in Figure 11). The web tuples in *temp*1 representing the old and new versions of the modified nodes are inserted in *updateTupleSet* as a single web tuple. For instance, the first web tuples in Figures 5(a) and (b) are contained in *temp*1. These two web tuples are combined together and inserted as a single web tuple (represented by the fourth web tuple in Figure 9). Similarly, the new and old versions of the web tuples in *temp*2 are determined and inserted in *updateTupleSet*. After Step (22) in Figure 11, the algorithm generates three sets, *insertTupleSet*, *deleteTupleSet* and *updateTupleSet* representing the web tuples that contains all the relevant nodes that are inserted, deleted or updated.

6.2.3 Algorithm of Phase 4

Next, the algorithm proceeds to create the delta web tables, i.e., W_{Δ^+}, W_{Δ^-} and W_{Δ^M} from *insertTupleSet*, *deleteTupleSet* and *updateTupleSet* (Steps (23) to (25) in Figure 11). We

describe the construction of Δ^M-web table here. As the construction of the Δ^+ and Δ^--web tables is straightforward, we do not discuss this in detail in this paper.

The algorithm first materializes each web tuple in *updateTupleSet* in the web tuple pool of W_{Δ^M}. Then, it retrieves the node objects (old and new versions) from the table node pools of W_1 and W_2 for each node id in *updateNodeSet* and materializes these nodes in the table node pool of W_{Δ^M}. Observe that in the table node pool of W_{Δ^M} we only materialize nodes which have undergone content modification. However, in web tuple pool of W_{Δ^M} we materialize the node ids of the joinable nodes in addition to the identifiers of the modified nodes. This is because each tuple in the web tuple pool contains not only the old and new versions of the modified nodes but also how these nodes are related to other nodes which have remained unchanged during t_1 and t_2. Finally, the schema of the joined web table and the outer joined web table are *manipulated* to generate the schema of W_{Δ^M}. As mentioned earlier, we do not discuss it in this paper.

The creation of Δ^+ and Δ^- web tables are quite similar to that of Δ^M-web table. The only difference is that the tuples are created from *insertTupleSet* and *deleteTupleSet* respectively and the schemas of W_{Δ^+} and W_{Δ^-} are identical to the web schema of W_1 or W_2.

7 Conclusions and Future Work

In this paper, we have formally defined the change detection problem in the web warehouse context. To solve this problem, we have presented algorithms that are based on representing two versions of Web data as web tables and manipulating these web tables using a set of web algebraic operators for detecting changes. We have represented the web deltas in the form of delta web tables. We have implemented algorithms for computing and representing changes in Web data relevant to a user's query.

As ongoing work, we are addressing the following issues: (1) Analytical and empirical studies of the algorithms for generating the delta web tables. We wish to perform experiments to evaluate the performance of the algorithms. We are also investigating the scalability issues in this context. (2) Currently, the delta web tables contain tuples where only some of the nodes represent the insertion, deletion or update operation during the transition. This is because we wish to show how these nodes are related to one another and to other nodes which have remained unchanged during the transition. Therefore, we need a mechanism to distinguish between the modified, new or deleted nodes in each delta web tables. We are currently building a data model over our warehouse data model to allow *annotation* on the affected nodes to represent these changes. (3) As we represent the web deltas in the form of web tables, these tables can be further manipulated

using existing set of web operators and queried. We are designing and implementing a powerful query language for the change management system in the context of our web warehouse. (4) We intend to implement a *change notification mechanism* in WHOWEDA similar to one proposed in [10]. We intend to support various subscription services such as allowing changes to be detected, queried, and reported whenever a user is interested. (5) Design an event-condition-action trigger language for WHOWEDA based on the ideas from the change detection system.

References

[1] URL-MINDER Web site. http://www.netmind.com/URL-minder/URL-minder.html.

[2] S. ABITEBOUL, D. QUASS, J. MCHUGH, J. WIDOM, J. WEINER. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries*, 1(1):68-88, April 1997.

[3] S. BHOWMICK, S. K. MADRIA, W.-K. NG, E.-P. LIM. Detecting and Representing Relevant Web Deltas Using Web Join. *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS'00)* , Taiwan, 2000.

[4] S. S. BHOWMICK. WHOM: A Data Model and Algebra for a Web Warehouse. *PhD Dissertation*, School of Computer Engineering, Nanyang Technological University, Singapore, 2001. Available at `www.ntu.edu.sg/home/assourav/` .

[5] S. S. BHOWMICK, W.-K. NG, S. MADRIA. Anatomy of a Coupling Query in a Web Warehouse. To appear in *International Journal of Software and Information Technology*, Elsevier Science, 2002.

[6] S. S. BHOWMICK, W.-K. NG, S. K. MADRIA . Schemas for Web Data: A Reverse Engineering Approach. *Data and Knowledge Engineering Journal (DKE)*, 39(2), pp. 105 – 142, Elsevier Science, 2001.

[7] S. BHOWMICK, S. K. MADRIA, W.-K. NG, E.-P. LIM. Web Warehousing: Design and Issues. *Proceedings of International Workshop on Data Warehousing and Data Mining (DWDM'98) (in conjunction with ER'98)*, Singapore, 1998.

[8] S. BHOWMICK, W.-K. NG, E.-P. LIM. Information Coupling in Web Databases. *Proceedings of the 17th International Conference on Conceptual Modeling (ER'98)*, Singapore, 1998.

[9] T. BRAY, J. PAOLI, C. SPERBERG-MCQUEEN. Extensible Markup Language (XML) 1.0. February 1998. W3C Recommendation available at `http://www.w3.org/TR/1998/REC-xml-19980210`.

[10] S. CHAWATHE, S. ABITEBOUL, J. WIDOM. Representing and Querying Changes in Semistructured Data. *Proceedings of ICDE 98*, Orlando, Florida, February 1998.

[11] Y-F. CHEN, F. DOUGLIS, HUALE HUANG, K. VO TopBlend: An Efficient Implementation of HtmlDiff in Java. *AT & T Labs - Research Technical Report*, 00.5.1, Available at `http://www.research.att.com/~ chen/topblend/`, January, 2000.

[12] YIH-FARN CHEN, GLENN S. FOWLER, ELEFTHERIOS KOUTSOFIOS, RYAN S. WALLACH. Ciao: A Graphical Navigator for Software and Document Repositories. *In Proceedings of International Conference on Software Maintenance*, pp. 66-75, 1995.

[13] S. CHAWATHE, HECTOR-GARCIA MOLINA. Meaningful Change Detection in Structured Data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*,Tuscon, Arizona 1997.

[14] S. Chawathe, A. Rajaraman et al. Change Detection in Hierarchically Structured Information. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Canada, June 1996.

[15] G. Cobena, S. Abiteboul, A. Marian. Detecting Changes in XML Documents. *Proceedings of the 18th International Conference on Data Engineering (ICDE' 2002)*, San Jose, California, 2002.

[16] F. Douglis, T. Ball, Y-F Chen, E. Koutsofios The AT & T Internet Difference Engine: Tracking and Viewing Changes on the Web. *World Wide Web Journal*, 1(1), pp. 27–44, January 1998.

[17] F. Douglis, T. Ball, Y-F Chen, E. Koutsofios WebGUIDE: Querying and Navigating Changes in Web Repositories. *Proceedings of Fifth International World Wide Web Conference*, Paris, May, 1996.

[18] D. S. Hirschberg. Algorithms for the Longest Common Sequence Problem. *Journal of the ACM*, 24(4):664-675, October 1977.

[19] G. Jacobson, Kiem-Phong Vo. Heaviest Increasing/Common Subsequence Problems. *Proceedings of the 3rd Annual Symp. of Combinatorial Pattern Matching*, Vol. 64, Springer-Verlag, pp. 52-65, 1992.

[20] L. Liu, C. Pu, W. Tang. WebCQ - Detecting and Delivering Information Changes on the Web. *Proceedings of the International Conference on Information and Knowledge Management (CIKM'00)*, Washington DC, USA, November, 2000.

[21] L. Liu, C. Pu, W. Tang. Continual Queries for Internet-scale Event-driven Information Delivery. *Special Issues on Web Technology. IEEE Transactions on Knowledge and Data Engineering (TKDE)*, March 1999.

[22] A. K. Luah, W.-K. Ng, E.-P. Lim. Locating Web Information Using Web Checkpoints. *Proceedings of the International Workshop on Internet Data Management (IDM'99)*, Florence, Italy, August 30-September 3, 1999.

[23] A. O. Mendelzon, G. A. Mihaila, T. Milo. Querying the World Wide Web. *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS'96)*, Miami, Florida,

[24] I. Mani, E. Bloedorn. Multi-document Summarization by Graph Search and Matching. Available at `http://www.mitre.org/support/papers/abstracts/multi_summariz.shtml`.

[25] D. Sasha, K. Zhang. Fast Algorithms for the Unit Cost Editing Distance Between Trees. *Journal of Algorithms*, 11:581-621, 1990.

[26] J. Widom, S. Ceri. Active Database Systems: Triggers and Rules for Advanced Database Processing.*Morgan Kaufmann*, San Fransisco, California, 1995.

[27] J. Tsong-Li Wang, K. Zhang, G. Chirn. Algorithms for Approximate Graph Matching. *Information Sciences*, 82(102):45–74, 1995.

[28] J. Tsong-Li Wang, B. Shapiro, D. Shasha, K. Zhang, K. Currey. An Algorithm for Finding the Largest Approximately Common Substructructures of Two Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 20, No. 8, pp. 889–895, August 1998.

[29] C. Yinyan, E. P. Lim, W. K. Ng. Storage Management of a Historical Web Warehousing System *Proceedings of the 11th International Conference on Database and Expert System Applications (DEXA'00)*, London, pp. 457–466, September, 2000.

[30] C. Yinyan. Querying Historical Web Information. *Master's Dissertation*, School of Computer Engineering, Nanyang Technological University, 2000.

Sourav S Bhowmick received his Ph.D. in Computer Engineering from Nanyang Technological University, Singapore in 2001. He is an Assistant Professor of the School of Computer Engineering at the Nanyang Technological University. He has published more than 40 journal and conference papers in the areas of web data management, bioinformatics and mobile data management. He is serving as PC member of various database conferences and workshops and reviewer for various database journals. He is a member of the ACM and IEEE Computer Society.

Sanjay Kumar Madria received his Ph.D. in Computer Science from Indian Institute of Technology, Delhi, India in 1995. He is an Assistant Professor of the Department of Computer Science at the University of Missouri-Rolla, USA. Earlier he was Visiting Assistant Professor in the Department of Computer Science, Purdue University, West Lafayette, USA. He has published more than 70 Journal and conference papers in the areas of web warehousing, mobile databases, data warehousing, nested transaction management and performance issues. He has chaired international conferences and workshops, organized tutorials, and has actively served in the program committees of numerous international conferences and has been reviewer for many journals. He participated as panelist in National Science Foundation and Swedish Research Council. He is an IEEE senior member and ACM member.

Wee Keong Ng is an Assistant Professor of the School of Computer Engineering at the Nanyang Technological University, Singapore. He obtained his M.Sc. and Ph.D. degrees from the University of Michigan, Ann Arbor in 1994 and 1996 respectively. He works and publishes widely in the areas of Web warehousing, information extraction, electronic commerce and data mining. He has organized and chaired international workshops, including tutorials, and has actively served in the program committees of numerous international conferences. He is a member of the ACM and IEEE Computer Society.