Abstract A Web and Document Database (WDDB) is a system to manage efficiently local documents and their semantic connection to remote ones. The general objective of a WDDB is to facilitate web search and internet navigation. Abstractly, a WDDB can be defined as a triple $<D, U, W>$, where $D$ stands for a local document database to store XML documents structurally, $U$ for a set of URLs with each pointing to a remote database which shares common data with the local one, and $W$ for a Web recognizer that identifies information sources related to data items in the local database. Then, a query against a WDDB normally consists of two parts: a local query and a set of remote queries. A local query can be considered as tree-embedding problem and can be sped-up using the so-called signature technique. A remote query has to be sent to another database which may not be available locally. To decide where to send a query, an address book has to be maintained, which can be established manually or automatically using Web recognizer.

Key Words: Web, Document databases, Tree inclusion, Signatures, Ontology, Path-oriented queries

1. Introduction

Recently, with the expansion of the Web, more and more comprehensive information repositories can be now visited easily through networks. A growing and challenging problem is how to quickly find information of interest to an individual in either a home or work setting. While navigating the Web, one may get lost in the maze of hyperlinks. A great deal of work has been done to mitigate this problem to some extent, including search engines such as Lycos, AltaVista, google and Yahoo, web query languages such as W3QL [20], semistructured data management systems [1, 33] and document databases [2, 5, 6, 7, 29]. However, these approaches lack a general method to bring together all the aspects such as the search engine, query treatment and document management under one umbrella. In this paper, we discuss a WDDB system to provide a powerful mechanism to guide the access of information sources distributed all over the world.

Abstractly, a WDDB can be defined as a triple $<D, U, W>$, where $D$ represents a local document database to store XML documents structurally, $U$ represents a set of URLs with each pointing to a remote database that shares some common data with the local one, and $W$ represents a Web recognizer that identifies information sources related to data items in the local database. More concretely, the remote information sources are established by storing the corresponding URLs, which are distributed over a predefined ontology. As an application scenario, consider a local database containing all the hotel information ($D$) in a city. Then, a query against it may get, for example, hotel prices, hotel living conditions, etc. But a user may also want to know about car rentals, sightseeing and different cuisine flavors in that city, which may be distributed in different databases. In this case, one has to switch over to those databases and submit new queries, respectively. However, if some URL links ($U$) are available and the relationships between them and the relevant local data items are specified, the system can manage to access those remote databases automatically. In addition, to obtain the URLs related to local data, a Web recognizer ($W$) is needed to explore the internet to find information sources of interest. Its other task would be to extract relevant information from the data obtained by issuing remote queries.

2. System Architecture

In terms of the discussion conducted in the introduction, we have the following system architecture for a WDDB.

part I - document management.

This part manages a local document database as an information source reachable over the network. Mainly, it contains:

1. A module for the schema management and the document loading. This module establishes a data schema for a given XML DTD and loads the corresponding documents into the database.
2. A module for query evaluation and
3. An interface that can be utilized for users to interact with the system.

part II - web connection.

This part is used to connect to remote document databases distributed over the internet. For this purpose, it contains:

Fig. 1. Architecture of a standalon WDDB
4. A module for web connection. In a local WDDB, a set of URLs is maintained and distributed over an ontology (see Section 4 for the definition of an ontology). That is, each concept (or a pattern) in the ontology is associated with a set of URLs pointing to remote document databases, which are related to the concept in some way. For example, for the ‘car rental’, we may have several URLs that are the addresses of some document databases containing the information on car rental enterprises. Therefore, a query involving a concept not available in the local resource can be sent to the associated remote document databases to get data for answering the query completely.

part III - web recognizer.

The third part is used to recognize remote information sources for a given concept. It mainly contains:

5. A module for web recognition, which can be done by establishing several patterns for a concept. These patterns can be utilized to find those document databases that contain XML pages matching any of them.

From the above system architecture, it can be seen that a WDDB always works together with some other document databases distributed over the network. All the relevant document databases are considered to be semantically connected through URLs, which are associated with a concept or a pattern in some ontology defined in the local WDDB.

3. Storage of documents in a WDDB

In a WDDB, documents are stored in a document database in XML format. It may be connected to remote document databases through URLs. In this section, we mainly discuss the storage of XML documents in a WDDB. The query evaluation is addressed in Section 4 in detail.

In Fig 2, we show a simple XML document, which contains element-tags, element-texts and attributes for elements. By means of the tags, the tree structure of the document is represented.

To keep the tree structure of documents when loading them into a relational database, we propose the following storage structure.

1. Element-tag:


   where DocID represents the document identifier, ID represents the element identifier, Tag is the element name (or tag name), firstChildID is the pointer to the first child of an element, siblingID is the pointer to the right sibling of an element and attributeID is the pointer to the first attribute of an element, which is stored in the relation Attribute.

2. Element-text:

   \{DocID: <integer>, textID: <integer>, value: <string>\},

   where textID is for the identifiers of texts that are the values of the corresponding elements in original documents. One should notice that a text always takes an element as its parent node, and if an element has a text as a child, it has only this child node. See the following table for illustration.

3. Attribute:

   \{DocID: <integer>, att-ID: <integer>, parentID: <integer>, att-name: <string>, att-value: <string>\}.

   In this relation, the attribute parentID is used for the identifiers of the corresponding elements (stored in relation Element-tag), with which an attribute is associated. The following table helps for a better understanding.

From the above discussion, we can see that the tree structure of a document is implemented through the attributes firstChildID and siblingID in the relation Element-tag, which contain pointers to the first child and the first right sibling node, respectively. They can be used to efficiently navigate documents and to support the checking for tree embedding, which is about to be discussed in the next section.

4 Query evaluation in a WDDB

In a WDDB, a query may be composed of two parts: a local query and a remote query. The local query can be evaluated against the local database while the remote query has to be sent to remote databases. In this section, we first give a general description of the WDDB’s queries in 4.1. Then, we discuss the evaluation of local queries and remote queries in 4.2 and 4.3, respectively.

4.1 Path-oriented queries

Several path-oriented language such as XQL [23] and XML-QL [14] have been proposed to manipulate tree-like structures as well as attributes and cross-references of XML documents. XQL is a natural extension to the XSL pattern syntax, providing a concise, understandable notation for pointing to specific elements and for searching nodes with particular characteristics. On the other hand, XML-QL has operations specific to data manipulation such as joins and supports transformations of XML data. XML-QL offers tree-browsing and tree-transformation operators to extract parts of documents to build new documents. XQL separates transformation operation from the query language. To make a transformation, an XQL query is performed first, then the
results of the XQL query are fed into XSL [31] to conduct transformation. Another interesting language is YATL [10]. It represents queries in a more compact form but has the same power as XQL and XML-QL. An interested reader is referred to [10] for more detailed description.

An XQL query is represented by a line command which connects element types using path operators (’/’ or ‘//’). ‘/’ is the branch operator which selects from immediate child nodes. ‘//’ is the descendant operator which selects from arbitrary descendant nodes. In addition, symbol ‘@’ precedes attribute names. By using these notations, all paths of tree representation can be expressed by element types, attributes, ‘/’ and ‘@’. Exactly, a simple path can be described by the following Backus-Naur Form:

```
<simple path>::=<PathOp> <SimplePathUnit> |<SimplePathUnit> |<ElementType>
<PathOp>::='/' | '//'
```

The following is a simple path-oriented query:

```
/letter/body [para $contains$'visited']
```

where /letter/body is a path and [para $contains$'visited'] is a predicate, enquiring whether element “para” contains a word ‘visited’.

Several paths can be joined together using ‘∧’ to form a complex query as follows.

```
/hotel-room-reservation/name ?x ∧
/hotel-room-reservation/location [city-or-district = 'Winnipeg'] ∧
/hotel-room-reservation/address [street = '510 Portage Ave.'] ∧
/car-rental/company/name ?y ∧
/car-rental/company/location [city-or-district = 'Winnipeg'] ∧
/car-rental/company/car-type ?z.
```

This query enquires the name of the hotel located at 510 Portage Ave., Winnipeg, as well as any car-rental company located in Winnipeg and any car types that are available in that company.

The above query can be represented in a compact form by integrating the common parts of multiple paths as shown below.

```
/hotel-room-reservation/[(name ?x ∧ location [city-or-district = 'Winnipeg'] ∧ street = '510 Portage Ave.')] ∧
/car-rental-company/[(name ?y ∧ location [city-or-district = 'Winnipeg'] ∧ car-type ?z)].
```

Assume that the local document database can answer the first part of the query. That is, it can provide the information on hotel room reservations, but fail to inform on car rentals. In this case, the system will send the second part of the query to some remote document databases pointed to by some URLs, which contain the information on the car rental. If one of the remote document databases is able to evaluate the query on car rentals, the answer will be sent back to the local WDDB, contributing to a complete answer to the original query.

A remote query can be of the form: <URL><query>. For instance, assume that there is a remote WDDB with the URL: http://www.uwinnipeg.ca/docDB, which contains the data on car-rental. The local database will issue a request of the following form to get the second part of the answer to the above query:

```
http://www.uwinnipeg.ca/docDB/car-rental/company/[name=$t/location/city-or-district=$x/car-type=$z].
```

The problem is how to determine where to send a remote query, and how a local WDDB becomes aware of other document databases and knows what they have. We discuss these issues in 4.3.

### 4.2 Evaluation of local queries

Both the documents and the queries can be considered as labeled trees and the evaluation of a local query can be thought of as a tree-embedding problem. In the following, we first define the concept of the tree embedding. Then, we show that for evaluating a query we will check whether a tree representing a query is embedded in another tree representing a document.

**Definition 1 (labeled tree)** A tree is called a labeled tree if a function label from the nodes of the tree to some alphabet is given, or say each node in the tree is labeled.

Obviously, an XML document can be represented as a tree with the internal nodes labeled with tags and the leaves labeled with texts. Similarly, a query as shown in 4.1 can also be represented as a tree labeled with tags and texts (or key words).

**Definition 2 (tree embedding)** Let $T_1$ and $T_2$ be two labeled trees. A mapping $M$ from the nodes of $T_2$ to the nodes of $T_1$ is an embedding of $T_2$ into $T_1$ if it preserves labels and ancestorship. That is, for any pair of nodes $u$ and $v$ of $T_2$, we require that:

1. $M(u) = M(v)$ if and only if $u = v$,
2. $label(u) = label(M(u))$, and
3. $u$ is an ancestor of $v$ in $T_2$ if and only if $M(u)$ is an ancestor of $M(v)$ in $T_1$.

Here, the mapping $M$ can be implemented as a method discussed in [24] or any method used in [26, 26, 4, 12].

**Example 1.** As an example, consider the trees: $T_1$ and $T_2$ shown in Fig. 4, representing the query shown discussed 4.1 and the document shown in Fig. 2, respectively. If a mapping as shown in Fig. 3 can be determined, we’ll have a tree-embedding of the tree representing the query into the tree representing the document as shown in Fig. 4.

```
M(T_1, hotel-room-reservation) = T_2.hotel-room-reservation
M(T_1, name) = T_2, name
M(T_1.location) = T_2.location
M(T_1, Travel-lodge) = T_2.515
```

For the query evaluation purpose, we’ll return that document as one of the answers.

In the following, we proposed a top-down algorithm for
checking whether $T_2$ is embedded in $T_1$, which needs $O(|T_1||T_2|)$ time and is worse than the bottom-up algorithm discussed in [8]. However, this top-down algorithm can be combined with the so-call signature technique to discard non-relevant document trees or subtrees as early as possible. Below we first describe this algorithm. Then, how to integrate signatures into the algorithm will be discussed in great detail.

![Fig. 4. Illustration for tree embedding](image)

The following algorithm checks whether a tree $T_1$ contains another tree (or forest) $T_2$.

**Algorithm tree-embedding($T_1$, $T_2$)**

**Input:** two trees: $T_1$ and $T_2$.

**Output:** if $T_1$ contains $T_2$, true; otherwise, false.

1. Let $r_1$ and $r_2$ be the roots of $T_1$ and $T_2$, respectively;
2. (*If $T_2$ is a forest, assume that it has a virtual root, which matches
any label.*)
3. if label($r_1$) = label($r_2$) then
   1. let $T'_{1i}$, ..., $T''_{1i}$ be the subtrees of $r_1$;
   2. let $T'_{2i}$, ..., $T''_{2i}$ be the subtrees of $r_2$;
   3. if there exist $i_1$, ..., $i_l$ such that $T_{1i_1} \lor ... \lor T_{1i_l}$ then return true;
   else if there exists some $i$ such that $T_{1i}$ matches $T_{2i}$ then return true;
   else if there exists some $i$ such that $T'_{1i}$ contains $T_{2i}$ then return true;
   else return false;
4. The algorithm works top-down. First, it checks whether the root $r_1$ of $T_1$ matches the root $r_2$ of $T_2$. If it is the case, all the subtrees of $r_2$ will be checked to see whether they are contained in the corresponding subtrees of $r_1$. If such containment cannot be achieved, the algorithm will check whether all the subtrees of $r_2$ is entirely contained in some subtree of $r_1$. If the root $r_1$ of $T_1$ does not match the root $r_2$ of $T_2$, we will check the containment of $T_2$ in a single subtree of $r_1$, too.

In the following, we discuss how to integrate the signature technique into a top-down tree embedding.

**Definition 3.** (Signature [15]) A signature for a key word is a hash-coded bit string of length $k$ with $m$ bit set to “1”, where $k$ and $m$ are determined according to the number of relevant documents in a database and the average number of key words in a document.

For example, using a hash function, we may assign three signatures with $k = 12$ and $m = 4$: 010 000 100 110, 100 010 010 100 and 010 100 011 000 to three key words: SGML, database and information, respectively.

**Definition 4.** (Document signature) Let $kw_1$, ..., $kw_n$ be the key words of a document. Let $s_i$ ($i = 1$, ..., $n$) be the signature for $kw_i$. Then, the signature $s$ for the document is set to be $s_1 \lor ... \lor s_n$.

Assume that a document has only three key words: SGML, database and information. Their signatures are shown as above. Then, the document signature: 110 110 111 110 can be obtained by superimposing the three signatures together (see Fig. 5).

![Fig. 5. Signature generation and comparison](image)

The goal of document signatures is to work as an inexact filter. Given a query $q$, we generate a signature $s_q$ for it using the same hash function as for documents. Then, we compare the query signature against the document signatures (which are stored in a signature file) and many nonqualifying signatures are discarded. The rest are checked using the tree-embedding algorithm. Three possible outcomes of the comparison are exemplified in Fig. 5: (1) the document matches the query; that is, for every bit set to 1 in $s_q$, the corresponding bit in the document signature $s$ is also set (i.e., $s \land s_q \neq s_q$) and the document contains really the query word; (2) the document doesn’t match the query (i.e., $s \land s_q = s_q$); and (3) the signature comparison indicates a match but the document in fact doesn’t match the search criteria (it is called a false drop). In order to eliminate false drops, the document must be examined after the document signature signifies a successful match. We do this by using a tree-embedding check.

The purpose of using a signature file is to screen out most of the nonqualifying documents. A signature failing to match the query signature guarantees that the corresponding document can be ignored. Therefore, unnecessary document accesses are prevented. Signature files have a much lower storage overhead and a simple file structure than inverted indexes [34].

To generate a query signature, we introduce the following concepts.

**Definition 5.** (Query tree) Let $P_1 \land ... \land P_n$ be a query $q$ with each $P_i$ ($i = 1$, ..., $n$) being of the form: $[p_{i1} / ... / p_{ik} \land \{p_{ij}.op < \text{value}\}]$, where each $p_{ij}$ is a tag, $op \in \{<, \leq, \geq\}$, and $\text{value}$ is a set of key words (i.e., $kw_1 \land ... \land kw_i$ or $kw_j \lor ... \lor kw_k$). Then, all the paths appearing in $q$ constitute a query tree, denoted $T_q$ (see Fig. 6(a) for illustration.)

**Definition 6.** (Query signature tree) Let $[p_{i1} ... / p_{ik} \land \{p_{ij}.op < \text{value}\}]$ be a path in $T_q$ (from the root to some leaf). Let $[p_{ij} / ... / p_{ij} \land \{p_{ij}.op < \text{value}\}]$
[\rho_{ij}, \text{op value}]$ be the corresponding subquery in $q$. Then, if \text{value} is of the form: $kw_1 \land \ldots \land kw_j$ then its signature $s_{\text{value}} = s_{kw_1} \lor \ldots \lor s_{kw_j}$, where each $s_{kw_i}$ represents the signature of $kw_i$. If \text{value} is of the form: $kw_1 \lor \ldots \lor kw_j$, then its signature $s_{\text{value}} = s_{kw_1} \land \ldots \land s_{kw_j}$. The query signature tree is denoted by $T_q$. The signature of a non-leaf node in $T_q$ can be obtained by superimposing the signatures of its child nodes.

For example, for the local part of our exemplary query, the query tree and the query signature tree are shown in Fig. 6(a) and (b), respectively.

![Fig. 6. Query tree and query signature](image)

Based on the above concepts, the evaluation of a query against a document database can be conducted using the following algorithm. Its inputs are a document tree and a query tree.

**Algorithm signature-tree-embedding($T_1, T_2$)**

input: two tree: $T_1$ and $T_2$

output: if $T_1$ contains $T_2$, true; otherwise, false.

1. let $s_1$ and $s_2$ be the signatures of $T_1$ and $T_2$, respectively;
2. if $s_1$ does not match $s_2$ then return false;
3. else
   1. let $r_1$ and $r_2$ be the roots of $T_1$ and $T_2$, respectively;
      1. if $\text{label}(r_1) = \text{label}(r_2)$ then
        1. let $T_1^1$, $\ldots$, $T_1^k$ be the subtrees of $r_1$;
        2. let $T_2^1$, $\ldots$, $T_2^j$ be the subtrees of $r_2$;
        3. if there exist $i_1$, $\ldots$, $i_j$ such that
           \[
           \prod (\text{tree-embedding}(T_1^{i_1}, T_2^{i_j}))
           \]
           then return true;
        4. else if there exists some $i$ such that
           \[
           \text{tree-embedding}(T_1^i, T_2)
           \]
           then return true;
        5. else return false;

The above algorithm is similar to the Algorithm tree-embedding($T_1, T_2$). The main difference is that at the very beginning, the signatures of $T_1$ and $T_2$ are compared to avoid useless searching. We notice that this optimization is recursively applied.

4.3 Database connection for remote query evaluation

In this subsection, we mainly address the database connection. It is necessary for evaluating a remote query. First, we discuss how to organize URLs in a local database in 4.3.1. Then, in 4.3.2, we discuss how a remote information source is recognized.

4.3.1. Web connection

As mentioned in the previous section, to evaluate a remote query, a WDDB has to know where to send that query. This can be done by maintaining a so-called association list of concepts. Each item of an association list is a triple of the form: $(G, C, S)$, where $G$ represents an information unit, e.g., some hotel information in a city, $C$ stands for a set of URLs connecting to some remote databases containing the relevant information such as car rental in that city, and $S$ is a descriptor of the relationship between $G$ and $C$. Assume that a document database contains only the information on hotel. Then, the query shown in 4.1 cannot be answered completely. However, using a corresponding item, say (’hotel’, [url1, url2, ..., urlj], ’car rental’) in the association list, the system can switch over to the document databases pointed to by url1, url2, ..., urlj to obtain the information on the car rental.

In an association list, a same concept may appear multiple times and some concepts are possibly closely related. To handle these issues, we organize the association list in a different way. We extend the concept of mediators discussed in [32], which is originally proposed to integrate heterogeneous information sources. Concretely, A mediator is composed of two parts: an ontology and a set of articulations. An ontology is a pair $(T, \subseteq)$, where $T$ is a set of names, or terms, and $\subseteq$ is a subsumption relation over $T$, i.e., a reflexive and transitive relation over $T$. If $a$ and $b$ are two terms of $T$, we say that $a$ is subsumed by $b$ if $a \subseteq b$; e.g., Database $\equiv$ Informatics, Canaries $\equiv$ Birds. An articulation is a set of relationships between the terms of the mediator and the terms of a local source. Through the articulations, the heterogeneity of local databases is suppressed.

For our purpose, a mediator in a WDDB is defined to be a tree-structure and a set of URLs. In the tree structure $\mathcal{G}$, a node is a pattern that is used to identify relevant information sources. Similar to $T$, an edge from $c$ to $d$ in $\mathcal{G}$ represents that the concept represented by $c$ subsumes the concept represented by $d$. Associated with $v$ (a node in $\mathcal{G}$), we have a set of URLs pointing to the web pages matching the pattern represented by $v$. As an example, consider the tree structure shown in Fig. 7.

![Fig. 7. Extended ontology](image)

Such a tree structure is called an extended ontology (EO for short) in the sense that a term in an ontology is extended to a more complex structure, i.e., a pattern to describe the concept more exactly. In an EO, a pattern is normally a tree to represent an information structure for a concept (see pattern2 in Fig. 7: it is used to recognize the pages for car rental). In the simplest case, a pattern can be a key word and in this case an EO is degenerated to a normal ontology. Such a pattern is used to find pages relevant to a concept from networks.

4.3.2 Web recognizer

To find the remote information sources related to a concept, we need a mechanism to recognize web pages. Normally, one can determine the similarity of two pages in different ways. For instance, one can use the information retrieval notion of textual similarity [24]. One could also use data mining techniques to cluster pages into groups that share meaningful terms (e.g., [22]), and then define pairs of pages within a cluster to be similar. A third option is to compute
textual overlap by counting the number of chunks of text (e.g., sentences or paragraphs) that pages share in common [25, 26, 4, 12]. In all these schemes, there is a threshold parameter that indicates how close pages must be to be considered similar (e.g., according to number of shared words, n-dimensional distance, number of overlapping chunks). This parameter needs to be empirically adjusted according to the target application.

All the methods mentioned above don’t, however, pay attention to an important aspect of information: the structure of a page. As we know, a page in HTML or XML format always consists of a hierarchical structure, starting with a root element as shown in Fig. 2.

Such structure information can be used to speed up page matchings (since taking the structure of pages into account can limit the search for similar terms to small parts of a text). A frequently used technique to explore the similarity of structures is tree matching; but it is too strict and a similar page may be filtered out undesirably. So we utilize the tree embedding technique once again for this task.

5. Conclusion

In this paper, we have discussed the system architecture of a WDDB, which is composed of three parts: a local document database, a web connector, and a web recognizer. The local document database can be considered as an information source reachable through the network. It can also connect to some other document databases through its web connector, which maintains a set of URLs. Each URL is related to a concept or a pattern that specifies the content of the document. The task of the web recognizer is to perform web recognition. It works as a web wrapper [3, 18] but is more powerful in the sense that it recognizes a web page by checking not only part of the page’s syntactic structure but the whole page with semantics considered. It will associate a set of URLs with a concept or a pattern which indicates the contents of the document databases pointed to by the URLs.

References

[9] Y. Chen, A New Way to Speed-up Recursion in Relational Databases, in: Proc. of 13th Information Resources Manage-