Web and Document Databases: an Effective Way to Explore the Internet

Yangjun Chen
Dept. Applied computer Science
University of Winnipeg, Winnipeg, Manitoba, Canada, R3B 2E9

***Abstract*—In this paper, we discuss the architecture of a system, the so-called *Web and Document Databases* (*WDDBS* for short), designed to explore the Internet effectively and efficiently. Abstractly, a *WDDBS* can be defined as a triple <$\mathcal{D}$, $\mathcal{P}$, $\mathcal{WS}$>, where (1) $\mathcal{D}$ stands for a local document database to store XML documents, (2) $\mathcal{P}$ for a subsystem responsible for remote query evaluation, including resolution of semantic conflicts among heterogeneous databases, and (3) $\mathcal{W}$ for a Web crawler which should be able to find information sources related to the local database in some way. Then, each information source can be organized into a *WDDB* distributed over the Internet, which may be connected to others through URLs. A query submitted to a WDDBS will first be evaluated against the local document database, and then possibly switched over to some remote document databases if necessary, which is controlled by the 'knowledge' on how local WDDBSs are connected. In this way, the load of traffic over the Internet can effectively be decreased, but the information explored is more relevant.**

Keywords: XML document; Web; tree pattern queries; semantic conflict resolution; hash tabels, signature trees.

I. INTRODUCTION

With the expansion of the Web, more and more comprehensive information repositories can be now visited easily through network. 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 hyper-links. A great deal of work has been done to mitigate this problem to some extent, including search engines such as *Yahoo*, *AltaVista* and *Google*, different web query languages such as W3QL [20], semistructured data management systems [1, 19] and document databases [18]. Our goal is to bring together all such mechanisms under one umbrella to guide the access of information resources distributed all over the world.

Abstractly, a *WDDBS* can be defined as a triple <$\mathcal{D}$, $\mathcal{P}$, $\mathcal{WS}$>, where $\mathcal{D}$ stands for a local document database to store XML documents, $\mathcal{P}$ for a subsystem responsible for remote query evaluation, including resolution of semantic conflicts among heterogeneous databases, and $\mathcal{W}$ for a Web crawler which should be able to find information sources related to some data items in the local database. Then, each information source can be organized into a *WDDBS* which may be connected to others through URLs. In an applied scenario, consider a local database containing all the hotel information ($\mathcal{D}$) in a city. Then, a query against it may get, for example, hotel prices, hotel living condition, etc. But a user may also want to know about auto rental, 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 to remote databases ($\mathcal{P}$) 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 a piece of local data, a Web crawler ($\mathcal{WS}$) is desired to explore the Internet to find information resources of interest. The other task of it would be to extract relevant information from the data obtained by issuing remote queries.

In Fig. 1, we give the architecture of WDDBS, showing how its main subcomponents are connected.

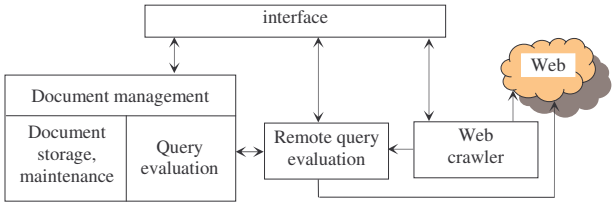

Fig. 1. WDDBS architecture

II. LOCAL DATABASES

The most important subcomponent of a WDDBS is the local XML database. It mainly contains three parts: document storage and maintenance, query evaluation, and integrity constrains, which are described below in detail.

A. Document storage and maintenance

There are different ways to store XML documents. A simple method is to decompose an XML document into elements and attributes and store them in four relations with the following structures:

DocRoot(*docID*, *rootElmentID*),
SubElement(*parentID*, *childID*, *position*),
ElementAttribute(*elementID*, *name*, *value*),
ElementValue(*elementID*, *value*)

However, it supports neither any efficient algorithm for evaluating tree pattern queries, nor any effective index mechanism. The reason for this is that the *parent-child*, as well as *ancestor-descendant* relationships cannot be efficiently manipulated. Although using the indexes over paths individual elements can be quickly located, it needs costly *path joins* to check tree matchings.

Another way is to store XML documents as data streams by using a kind of tree encoding, which can be used to

identify different relationships between the nodes of a tree.

Let T be a document tree. We associate each node v in T with a quadruple (d, l, r, ln), denoted as $\alpha(v)$, where d = DocId, l = LeftPos, r = RightPos, and ln = LevelNum, defined to be the nesting depth of the element in the document. (See Fig. 2 for illustration.) By using such a data structure, the structural relationships between the nodes in an XML database can be simply determined [22]:

(i) *ancestor-descendant*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is an ancestor of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$.

(ii) *parent-child*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is the parent of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $l_1 < l_2$, $r_1 > r_2$, and $ln_2 = ln_1 + 1$.

(iii) *left-to-right order*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is to the left of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $r_1 < l_2$.

In Fig. 2, v_2 is an ancestor of v_6 and we have v_2.LeftPos = 2 < v_6.LeftPos = 6 and v_2.RightPos = 9 > v_6.RightPos = 6. In the same way, we can verify all the other relationships of the nodes in the tree. In addition, for each leaf node v, we set v.LeftPos = v.RightPos for simplicity, which still work without downgrading the ability of this mechanism. In the rest of the paper, if for two quadruples $\alpha_1 = (d_1, l_1, r_1, ln_1)$ and $\alpha_2 = (d_2, l_2, r_2, ln_2)$, we have $d_1 = d_2$, $l_1 \le l_2$, and $r_1 \ge r_2$, we say that α_2 is subsumed by α_1. For convenience, a quadruple is considered to be subsumed by itself (i.e., a node is considered to be an ancestor of itself). In this way, we can store an XML document as several streams of quadruples with each associated with a different tag name and sorted by LeftPos or RightPos values.

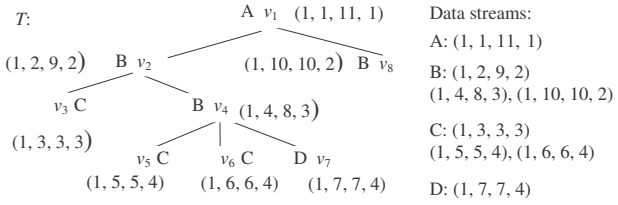

Fig. 2. Illustration for tree encoding and data

If no confusion is caused, we will used v and $\alpha(v)$ interchangeably. Also, as with *DeweyIDs*, we can leave gaps in the numbering space between consecutive labels to support dynamical changes of documents.

B. Query evaluation

In order to enquire an XML database, we use a language such as *XQuery*, *XML-QL*, or *Quilt*.

Analogous to *SQL* select-from-where expressions, XQuery provides an *FLWR* structure to specify queries, as illustrated in the following example:

```
let   $p := doc("publication.xml"), $q := $p//AuthorBook
for   $s in $q[.//Author/@name = 'D. Knuth']//Book/Title
where $q//Book/Title[@year = '1973']
return $s.
```

Special attention should be paid to the *for*-clause in it, which is an XPath to represent, together with the where-clause, a searching condition. Such an searching condition typically specify patterns of selection predicates on multiple elements that have some tree-structured relationships. For instance, the searching condition in the above FLWR expression can be represents as a tree structure shown in Fig. 3.

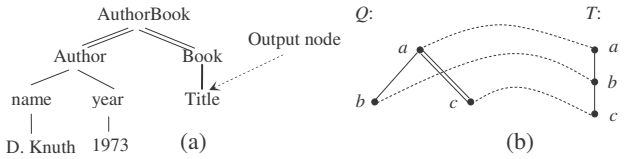

Fig. 3. A query tree and a tree matching a path

Therefore, to answer a query, we need to find all occurrences of a tree pattern in a database. It is the so-called tree matching problem, for which different strategies for tree matching have been developed.

Tree matching

From the above discussion, we can see that to evaluate an XQuery query we need to do a tree matching. Formally, a tree matching is defined as follows.

Definition 1 An embedding of a tree pattern Q into an XML document T is a mapping f: $Q \to T$, from the nodes of Q to the nodes of T, which satisfies the following conditions:

(i) Preserve node label: For each $u \in Q$, $label(u) = label(f(u))$ (or say, u matches $f(u)$).

(ii) Preserve *parent-child/ancestor-descendant* relationship: If $u \to v$ in Q, then $f(v)$ is a child of $f(u)$ in T; if $u \Rightarrow v$ in Q, then $f(v)$ is a descendant of $f(u)$ in T.

If there exists a mapping from Q into T, we say, Q can be imbedded into T, or say, T contains Q.

Almost all the existing strategies for evaluating tree patterns queries are designed according to this definition [2, 3, 5, 7, 13, 14, 15, 18], which can roughly be divided into two categories. One is based on path indexes, and the other is based on the *XB-tree* structure. For example, the methods discussed in [18] are typically path-index-based, by which a document is decomposed into a set of binary relationships between pairs of nodes, such as parent-child and ancestor-descendant relations, or into a set of paths. The sizes of intermediate relations tend to be very large, even when the input and final result sizes are much more manageable. To make the matter worse, path joins are needed, which requires exponential *cpu* time in the worst case.

The methods discussed in [2, 3, 4, 5] are all based on the XB-tree structure, which run in polynomial time and needs only a space linear in the size of documents.

Definition 1 allows a path to match a tree as illustrated in Fig. 3(b). It is because by Definition 1 the left-to-right relationships between siblings are not taken into account. We call such a problem an *unordered tree pattern matching*.

We may consider another problem, called an *ordered tree pattern matching*, defined below.

Definition 2 An embedding of a tree pattern Q into an XML document T is a mapping f: $Q \to T$, from the nodes of Q to the nodes of T, which satisfies the following conditions:

(i) same as (i) in Definition 1.

(ii) same as (ii) in Definition 1.

(iii) Preserve *left-to-right order*: For any two nodes $v_1 \in Q$ and $v_2 \in Q$, if v_1 is to the left of v_2, then $f(v_1)$ is to the left of $f(v_2)$ in T.

In general, a node u_1 is said to be to the left of another node u_2 in a tree T if they are not related by the ancestor-descendant relationship and u_2 follows u_1 when we traverse T in preorder.

This kind of tree mappings is useful in practice. For example, an XML data model was proposed by Catherine and Bird [6] for representing interlinear text for linguistic applications, used to demonstrate various linguistic principles in different languages. For the purpose of linguistic analysis, it is essential to preserve the linear order between the words in a text [6]. In addition to interlinear text, the syntactic structure of textual data should be considered, which breaks a sentence into syntactic units such as noun clauses, verb phrases, adjectives, and so on. These are used by the language *TreeBank* to provide a hierarchical representation of sentences. Therefore, by the evaluation of a tree pattern query against the TreeBank, the order between siblings should be considered.

The method discussed in [4] needs only polynomial time for this problem and uses the XB-tree as its indexing structure. (The method described in [16] is also for the ordered tree matching, using a *trie* as the indexing structure. But its time complexity is exponential in the size of query nodes.)

In the following, we discuss an enhanced version of XB-trees with an extra ability to cut off irrelevant documents by using the so-call signatures technique.

XB-tree – an efficient index structure

An *XB-tree* [2, 4] over an XML data stream is just a modification of the well-known B^+*-tree* indexing structure, as illustrated in Fig. 4(a), which is an XB-tree built over the data stream shown in Fig. 4(b).

Each entry in a page (a node) P of an XB-tree consists of a bounding segment [LeftPos, RightPos] and a pointer to its child page, which contains entries with bounding segments completely included in [LeftPos, RightPos]. The bounding segments may partially overlap, but their LeftPos positions are in increasing order. Besides, each page has two extra data fields: *P.parent* and *P.parentIndex*. *P.parent* is a pointer to the parent of P, and *P.parentIndex* is a number i to indicate that the ith pointer in *P.parent* points to P. For instance, in the XB-tree shown in Fig. 4(b), P_3.*parentIndex* = 2 since the second pointer in P_1 (the parent of P_3) points to P_3.

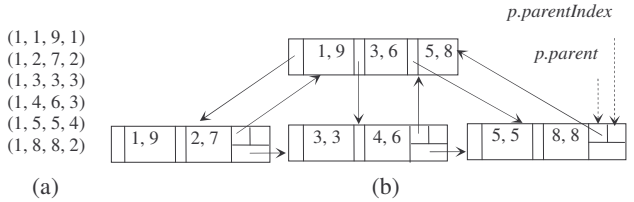

Fig. 4. A quadruple sequence and the XB-tree over it

By a method which uses XB-trees as indexes, each node q in a query Q will be associated with a data stream, denoted $B(q)$, such that for each $v \in B(q)$ $label(q) = label(v)$. Over such a data stream, an XB-tree may be constructed. We notice that in a Q we may have more than one query nodes q_1, ..., q_k with the same label. So they will share the same data stream and then the same XB-tree. For each q_j (j = 1, ..., k), we maintain a pair (P, i), denoted β_{q_j}, to indicate that the ith entry in the page P is currently accessed for q_j. Thus, each (j = 1, ..., k) corresponds to a different searching of the same XB-tree as if we have a separate copy of that XB-tree over $B(q_j)$.

In [2], two operations are defined to navigate an XB-tree, which change the value of β_q.

1. *advance*(β_q) (going up from a page to its parent): If $\beta_q = (P, i)$ does not point to the last entry of P, $i \leftarrow i + 1$. Otherwise, $\beta_q \leftarrow$ (*P.parent*, *P.parentIndex* + 1).
2. *drilldown*(β_q) (going down from a page to one of its children): If $\beta_q = (P, i)$ and P is not a leaf page, $\beta_q \leftarrow (P', 1)$, where P' is the ith child page of P.

Initially, for each q, β_q points to (*rootPage*, 0), the first entry in the root page. We finish a traversal of the XB-tree for q when β_q = (*rootPage*, *last*), where *last* points to the last entry in the root page, and we advance it (in this case, we set β_q to ϕ, showing that the XB-tree over $B(q)$ is exhausted.) As with *TwigStackXB* [2], the entries in $B(q)$'s will be taken from the corresponding XB-tree; and many entries can possibly be skipped Each time we determine a q ($\in Q$), for which an entry from $B(q)$ is taken, the following three conditions are satisfied:

i) For q, there exists an entry v_q in $B(q)$ such that it has a descendant in each of the streams $B(q_i)$ (where q_i is a child of q.)
ii) Each recursively satisfies (i).
iii) LeftPos(v_q) is minimum.

To determine which XB-tree will be accessed in a next step, we use the function *getNext*( ) given in [2], in which the following functions are used.

isLeaf(q) - returns *true* if q is a leaf of Q; otherwise, *false*.

isRoot(q) - returns *true* if q is the root of Q; otherwise, *false*.

currL(β_q) - returns the LeftPos of the entry pointed to by β_q.

currR(β_q) - returns the RightPos of the entry pointed to by β_q.

isPlainValue(β_q) - returns *true* if β_q is pointing to a leaf node in the corresponding XB-tree.

end(Q) - if for each leaf node q of Q $\beta_q = \phi$ (*i.e.*, $B(q)$ is exhausted), then returns *true*; otherwise, *false*.

Function *getNext*(q) (*Initially, q is the root of Q.*)
begin
1. **if** (*isLeaf*(q)) **then** return q;
2. **for** each child q_i of q **do**
3. {$r_i \leftarrow$ *getNext*(q_i);
4. **if** ($r_i \neq q_i \vee \neg$*isPlainValue*(β_q)) **then** return q;}
5. $q_{min} \leftarrow q'$ such that *currL*($\beta_{q'}$) = min_i{*currL*(β_{q_i})};
6. $q_{max} \leftarrow q''$ such that *currL*($\beta_{q''}$) = max_i{*currL*(β_{q_i})};
7. **while** (*currR*(β_q) < *currL*($\beta_{q_{min}}$) **do** *advance*(β_q);

8. **if** ($currL(\beta_q) < currL(\beta_{q_{max}})$) **then** return q;

9. **else** return q_{min}; }

end

The goal of the above function is to figure out a query node to determine what entry from data streams will be checked in a next step, which has to satisfy the above conditions (i) - (iii). Lines 7 - 9 are used to find a query node satisfying condition (i) (see Fig. 5 for illustration of line 7.) The recursive call performed in line 3 shows that condition (ii) is met. Since each XB-tree is navigated top-down and the entries in each node is scanned from left to right, condition (iii) must be always satisfied.

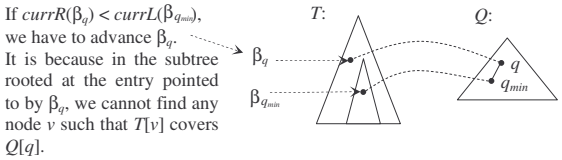

Fig. 5. Illustration for *advance*(β_q)

However, an XB-tree possesses no filtering mechanism to discard irrelevant documents as early as possible, which greatly delays response time. To address this problem, we will integrate the so-called *signature file* technique [8] into it to cut off irrelevant data. Intuitively, a signature for a key word is a hash-coded bit string of length m with k bits set to 1 ($k < m$). Then, a signature for a single document can be created by superimposing together the signatures for all the key words appearing in it. (By 'superimposing' we mean a bitwise OR operation.) When a query arrives, a query signature can be generated by applying the same hash function, and used to discard irrelevant documents according to the following rules: (i) the document signature s matches the query signature s_q; that is, for every bit set in s_q , the corresponding bit in the document signature s is also set (i.e., $s \wedge s_q = s_q$) and the document really contains the query words; (ii) the document does not match the query (i.e., $s \wedge s_q \neq s_q$) and therefore can be discarded; and (iii) the signature comparison indicates a match but the document in fact does not match the search criteria (referred to as a *false drop*); so the document itself needs to be checked.

Fig. 6 depicts the signature generation and comparison process of a document containing three key words: "John", "12345678", and "professor".

Fig. 6 depicts generation of a document signature and comparison process of an object having three attribute values: "John", "12345678", and "professor".

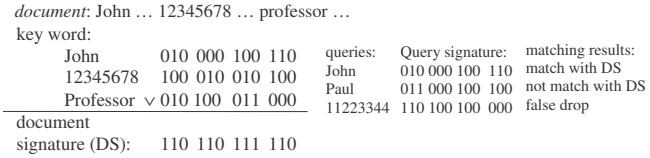

Fig. 6. Document signature generation and comparison

In order to equip an XB-tree with the ability of filtering irrelevant documents, we will associate each entry in a non-leaf node of the XB-tree with a signature as illustrated in Fig. 7.

We assume that the XB-tree shown in Fig. 7 is built over a data stream for tag name '*name*'. Then, each entry in a leaf node is associated with an author name, for which a signature can be created using a hash function. Superimposing all the signatures in a leaf node, we will generate a signature for the corresponding entry in its parent node. In the same way, we can superimposing all the signatures in the parent to create a new signature and put in a higher-level node, and so on.

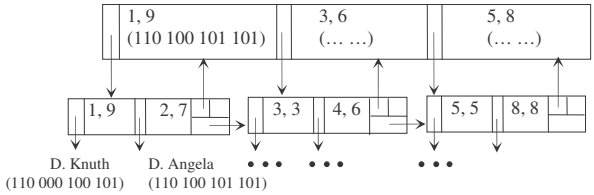

Fig. 7. Integration of signatures into XB-trees

Thus, when searching an XB-tree, we can also use signatures to skip many nodes and discard irrelevant documents.

C. Integrity constraints

As in a relational database, integrity constrains can be specified in an XML database to keep data consistence. In addition, it can also be utilized to speed up query evaluation. For example, consider a query '*find the title and author of books that have a publisher*'. If we have specified a constraint such as '*every book has a publisher*'. Then, the query can be simplified to '*find the title and author of books*.' In our research, four kinds of constraints are recognized: (i) *co-occurrance*: types A and B always occur together as children of another type, denoted by $A \downarrow B$. (ii) *subtype*: every document node of type A is also of type B, denoted by $A \leq B$. For example, in a document, there may exist some nodes labeled with the type "technician" while some other nodes with the type "employee". Obviously, we have "technician" ≤ "employee". (iii) *required child*: every document node of type A has a child of type B, denoted by $A \rightarrow B$. (vi) *required descendant*: every document node of type A has a descendant of type B, denoted by $A \Rightarrow B$.

The goal of our research on the integrity constraints is to use them to simplify queries, and cut off searching space as well.

D. Some other important issues

In this section, we discuss some other important issues, such as *IDREF/ID* links and XPaths with complicated predicates.

IDREF/ID links

In an XML document, we can associate a set of attributes with an element. Especially, we can assign an identifier to it as an attribute value (referred to as an *ID* attribute), which can be referenced by another element by using this identifier as one of its attribute values (referred to as an *IDREF* attribute). In this sense, an XML document is

considered to be a sparse directed graph: a tree plus some *IDREF/ID* links. Thus, in some cases, the tree matching method cannot be used to evaluate a query efficiently. To know this clearly, let's have a look at the following XPath expression:

/AuthorBook[.//Author/name = 'D. Knuth' ∧
.//Author/@AuthorID = .//Book/@authorOf]/ Book/Title,

where @AuthorID refers to the *ID* attribute of element *Author* while @authorOf the *IDREF* attribute of element *Book*. This query asks for all the books authored by D. Knuth. However, it cannot be represented by a tree, but by a graph as shown in Fig. 8.

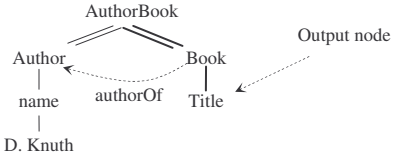

Fig. 8. A query graph

Therefore, the evaluation of such kind of queries is a process to check *subgraph isomorphism*, which is in general *NP-hard*. But a document graph is a special kind of graphs; a tree plus *IDREF/ID* links. So we can first do a tree matching and then check links, which needs only polynomial time. For instance, the above XPath expression can be rewritten as follows:

(/AuthorBook[.//Author/name = 'D. Knuth' ∧
.//Author/@AuthorID = x]/ Book[.//Book/@authorOf = y]/Title) ∧ (x = y).

In the above expression, the first part can be represented as a tree. Thus, it can be evaluated by using any strategy for this task. Then, for each answer obtained, we will check the values respectively for x and y to see whether they are equal.

General XPath expressions)

In a general XPath expression, predicates (such as 'name = D. Knuth') can be connected by both ∧ and ∨. One can even use a negated predicate (such as '¬(name = D. Knuth)') in an expression. Our initial idea is to decompose an expression into several sub-expressions such that each of them only contains ∧. Then, each of them will separately be evaluated. Unifying their results, we will get the final answer. For example, an expression given below

Book[Title = 'XML']//Author[name = 'Jane' ∨
name = 'Doe']

can be transformed into two sub-expressions:

Book[Title = 'XML']//Author[name = 'Jane'],
Book[Title = 'XML']//Author[name = 'Doe'].

III. Remote Query Evaluation

In order to access remote databases when necessary, we maintain a matrix M with each entry $M(i, j)$ being a structure: <*descriptor*; url_a, url_b> to record how the databases DB_i and DB_j are connected, where url_a and url_b are the URL addresses of DB_i and DB_j, respectively. If DB_i and DB_j are homogeneous, *descriptor* is simply a label (or a word) to indicate their relationship. For example, the descriptor for DB_{hotel} and $DB_{auto\text{-}rental}$ can be *tourism*. However, if DB_i and DB_j are heterogeneous, *descriptor* will be a complex structure which provides a way to resolve the semantic conflict between DB_i and DB_j. Concretely, it is a structure containing three parts: ontology for DB_i, ontology for DB_j, and a mapping between the two ontologies.

In general, an ontology can be represented as a graph, in which each node represents a concept or a relation, and an edge from node v to node u represents one of four relationships: v is a subclass of u, v is a subproperty of u, v is the domain of u, and v is the range of u. Such an ontology can be stored as an XML document following RDF/XML syntax, where RDF (Resource Description Framework) is a general-purpose language, developed by the W3C for representing information in the Web [17, 21].

The mapping between two ontologies can be established by using the so-called *Description Logics* (*DL*) [9], designed for representing knowledge and reasoning about it. Besides elementary descriptions for atomic concepts, atomic roles (properties or relationships), universal concept (⊤) and bottom concept (⊥), *DL* also provides four mapping assertions: equivalence, overlapping, disjoint, and subsumption (a concept C_1 is subsumed by another concept C_2 if the instances described by C_2 can be described by C_1 but the inverse is not true).

A WDDBS will provide all the above described functionalities. In addition, we will extend *DL* with a new mapping relation: *derivation* to indicate a concept in an ontology can be derived from some concepts in another ontology. For example, we may specify a mapping as shown below:

$Ontology_1$(father, brother) → $Ontology_2$(uncle).

Such kind of mappings enables us to accommodate more heterogeneity [10].

IV. Web Crawler

As in any web search engine, WDDBS uses a web crawler to explore the internet to find web pages of interests or relevant document databases distributed all over the world. Theoretically, a crawler can be a single machine that is started with a set S, containing the URL's of one or more web pages to crawl. There is a repository R of pages with the URL's that have already been crawled. Initially, R is empty. In order to check whether a new page has already been in R at each step, a search engine typically maintains a hash table H containing all the hash-coded signatures for all the pages stored in R. Each time a page is found, the hash-coded signature for it will be created and checked whether it is in H. We will change this process by maintaining a signature tree [8] for all the pages in R, instead of a hash table. In this way, we will replace a hash table searching with a signature tree searching, which should be a much more efficient process.

As an example, consider a set H of signatures shown in Fig. 9(a). We can store it as a tree structure as shown in Fig. 9(b). Such a tree has the following properties [8]:

i) Each node is labeled with a number $1 \le i \le l$, where l is the length of a signature in the hash table.
ii) Each left edge going out of a node is labeled with 0. Each right edge 1.
iii) Each path from the root to a leaf node corresponds to the identifier of a signature s_k in H, which is represented by a pair sequence: $(i_1, s_k[i_1]) \ldots (i_h, s_k[i_h])$ such that the jth node on the path is labeled with i_j and the jth edge labeled with $s_k[i_j]$, where $s_k[i]$ is the ith bit of s_k. There exists no p ($\neq k$) such that $(i_1, s_p[i_1]) \ldots (i_h, s_p[i_h]) = (i_1, s_k[i_1]) \ldots (i_h, s_k[i_h])$.

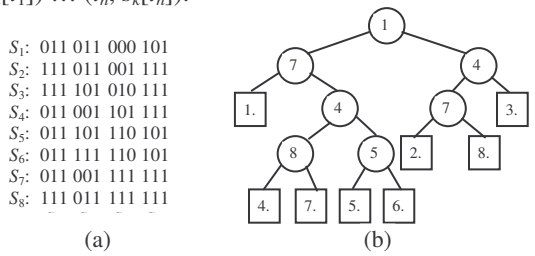

Fig. 9. A hash table and a signature tree

Due to the address collision, the time for searching a hash table may be very long. But searching a signature tree needs only O(l) time.

Another important issue concerning the web crawler is the page ranking. The current mechanism used in a search engine is based on the concept of 'page importance', which is calculated by counting the incoming links to a page. Concretely, the following recursive equation (established according to the theory of *Markov Chain*) will be used to estimate page importance [12]:

$$\boldsymbol{P} = (1 - \beta)\boldsymbol{N}\boldsymbol{P} + \beta\boldsymbol{J},$$

where $\boldsymbol{N}$ is a $k \times k$ transition matrix with $\boldsymbol{N}(i, j) = 1/r$ if page j has a link to page i, and there are a total $r \ge 1$ pages that j links to; otherwise, $\boldsymbol{N}(i, j) = 0$. $\boldsymbol{P}$ is a vector of k fraction numbers with each representing a page-importance to be determined. $\boldsymbol{J}$ is also a vector but with initial values set by the system to escape *spider traps*, each of which is a set of pages without outgoing links, and β is the probability that a spider trap appears.

The problem of the above method consists in the big size of $\boldsymbol{N}$. So we need to store $\boldsymbol{N}$ in a compact way. For example, we can store $\boldsymbol{N}$ as a graph with each edge associated with an entry in $\boldsymbol{N}$, and relax the exact solution to the equation to an approximate one by simulating random walk processes [11].

V. Conclusion

In this paper, the architecture of WDDBS is discussed, which is designed to facilitate the internet navigation. The system mainly contains three parts: local XML document management, remote query evaluation, and web crawling. In the local document system, a set of XML documents is stored and maintained, including query evaluation, indexing, and specification of integrity constraints. In the subsystem for remote query evaluation, a set of ontologies for different connected document databases, as well as their mappings, is maintained to resolve semantic conflicts. Finally, the web crawler is used to explore the internet to find information of interest, or all those document databases connected to the local one in some way.

References

[1] A. Bonifati, and S. Ceri, Comparative analysis of five XML query languages, *SIGMOD Record*, 29(1), 2000, pp. 68-79.

[2] N. Bruno, N. Koudas, and D. Srivastava, Holistic Twig Joins: Optimal XML Pattern Matching, in *Proc. SIGMOD Int. Conf. on Management of Data*, Madison, Wisconsin, June 2002, pp. 310-321.

[3] S. Chen, H-G. Li, J. Tatemura, W-P. Hsiung, D. Agrawa, and K.S. Canda, *Twig^{2}Stack*: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents, in *Proc. VLDB*, Seoul, Korea, Sept. 2006, pp. 283-294.

[4] Y. Chen and Y.B. Chen, Tree Reconstruction and Bottom-up Evaluation of Tree Pattern Queries, accepted by *Int. Conf. on Information Science and Applications (ICISA 2010)*, Seoul, Korea.

[5] Y. Chen, Bottom-up Evaluation of Twig Join Queries in XML Document Databases, in *Proc. 20th International Conf. on Database and Expert Systems Application (DEXA'09)*, Springer Verlag, Linz, Austria, August 31 - Sept. 4, 2009, pp. 356-363.

[6] B. Catherine and S. Bird, Towards a general model of Interlinear text, in *Proc. of EMELD Workshop*, Lansing, MI, 2003.

[7] Y. Chen and Y.B. Chen, A New Tree Inclusion Algorithm, *Information Processing Letters* 98(2006) 253-262, Elsevier Science B.V.

[8] Y. Chen and Y.B. Chen, On the Signature Tree Construction and Analysis, *IEEE Transaction on Knowledge and Data Engineering*, Vol. 18, No. 9, Sept. 2006, pp. 573-596.

[9] F. Baader, I. Horrocks, U. Sattler, Description Logics as Ontology Languages for Semantic Web, *Lecture Notes in Artificial Intelligence*, Springer Verlag, 2003.

[10] Y. Chen, A Systematic Method for Query Evaluation in Distributed Heterogeneous Relational Databases, *Journal of Information Science and Engineering*, Vol. 16, No. 4, 2000, pp. 463-497.

[11] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós, Towards Scaling Fully Personalized PageRank: Algorithms, Lower Bounds, and Experiments, Internet Mathematics Vol. 2, No. 3: 333-358.

[12] H. Garcia-Molina, J. Ullman, and J. Widdom, Database Systems: The Complete Book, Prentice Hall, 2008.

[13] G. Gottlob, C. Koch, and K.U. Schulz, Conjunctive queries over trees, *ACM PODS'2004*, pop. 189-200.

[14] G. Gou and R. Chirkova, Efficient Algorithms for Evaluating XPath over Streams, in: *Proc. SIGMOD*, June 12-14, 2007.

[15] P. Ramanan, Holistic Join for Generalized Tree Patterns, *Information Systems* 32 (2007) 1018-1036.

[16] P. Rao and B. Moon, Sequencing XML Data and Query Twigs for Fast Pattern Matching, *ACM Transaction on Database Systems*, Vol. 31, No. 1, March 2006, pp. 299-345.

[17] RDF Semantics. http://www.w3.org/TR/ref-mt.

[18] C. Seo, S. Lee, and H. Kim, An Efficient Index Technique for XML Documents Using RDBMS, *Information and Software Technology* 45(2003) 11-22, Elsevier Science B.V.

[19] K. Wang and H. Liu, Discovering structural association of semistructured data, *IEEE transaction on knowledge and data engineering*, Vol. 12, No. 3, May/June 2000, pp. 353-371.

[20] World Wide Web Consortium. XQuery 1.0: An XML Query Language, W3C Recommendation, Version 1.0, Jan. 2007. See http://www.w3.org/TR/xquery.

[21] Z. Wu et al., Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. in Proc. *24th Int. Conf. on Data Engineering (ICDE 2008)*, IEEE, April 2008, pp. 1239-1248.

[22] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman, on Supporting containment queries in relational database management systems, in *Proc. of ACM SIGMOD*, 2001.