SECURING XML DOCUMENTS

Charles A. Shoniregun
School of Computing and Technology,
University of East London,
Longbridge Road, Barking, Dagenham Essex, RM8 2AS, UK
E-mail: C.Shoniregun@uel.ac.uk,
cshoniregun@acm.org,
C.Shoniregun@infonomis.org.uk

Oleksandr Logvynovskiy
Business, Computing & Information Management,
London South Bank University,
Borough Road, London SE1 OLU UK
E-mail: a.logvynovskiy@lsbu.ac.uk,
a.logvynovskiy@infonomis.org.uk

ABSTRACT

XML (extensible markup language) is becoming the current standard for establishing interoperability on the Web. XML data are self-descriptive and syntax-extensible; this makes it very suitable for representation and exchange of semi-structured data, and allows users to define new elements for their specific applications. As a result, the number of documents incorporating this standard is continuously increasing over the Web. The processing of XML documents may require a traversal of all document structure and therefore, the cost could be very high. A strong demand for a means of efficient and effective XML processing has posed a new challenge for the database world. This paper discusses a fast and efficient indexing technique for XML documents, and introduces the XML graph numbering scheme. It can be used for indexing and securing graph structure of XML documents. This technique provides an efficient method to speed up XML data processing. Furthermore, the paper explores the classification of existing methods impact of query processing, and indexing.

Keyword: XML data, XML document, numbering scheme, securing graph structure

INTRODUCTION

The web and database communities have developed several approaches to resolve this problem. One is to convert semi-structured data into structured. This requires a pre-defined schema and a non-trivial translation of the data. Another option uses a partial schema to convert the XML data into structured data, where a partial schema is extracted from the data by using data mining techniques. Data that does not fit into this schema, will be stored and queried separately.
An XML query combines searching by value and structure. The structure search incorporates navigating through relationships among the elements. The major difficulties of the structure search are full XML document traversals and presence of cross-references that may require a closure computation. As the costs of these operations are significantly high, it is especially important to employ suitable indexes to speed up the searching operations. The processing of XML documents may require a traversal of all document structure and, therefore, the cost could be very high (Logvynovskiy and Lü, 2003). A strong demand for a means of efficient and effective XML processing has posed a new challenge for the database world.

RELATED WORK

Within XML databases there are usually two kinds of indexes: value (data) indexes and structure (path) indexes. Value indexes deal with coercing values and types of elements while structure indexes are meant to provide access to nodes reachable through certain paths. The LORE project uses four indexes: two for value and two for structure. The structure indexes were used to simplify searching of paths of elements and provide access to parents of nodes. The value indexes were designed for type coercion. LORE also uses a structure called DataGuide as a path index. DataGuide (or strong DataGuide) is a concise labelled graph representing each path in the data exactly once. DataGuide is used for query optimisation and as a path index, though it requires several index lookups (McHugh, 1999), while the STORED system applies data mining techniques to discover a

partial schema and then converts semistructured data into relations. For the parts of the semistructured data that do not fit the schema are stored separately in native form. Cooper et al (2001) have considered an approach of indexing XML paths as strings. They use a structure, called Index Fabric, analogous to a B-tree, to store paths encoded as stings. This technique has been implemented over relational database management systems (DBMSs) and has shown better results than traditional indexing schemes.

Li and Moon (2001) proposed the XISS system to index and store XML data. There are two indexes used for value related data and three indexes used for structure related data. These three indexes (value / structure related data) are element, attribute, and structure indexes. Along with the identifier of each node these indexes contain a pair of numbers assigned to the node – extended traversal number and size of the node, which are later used by a selection algorithm for computing of ascendant-descendant relationship without tree traversal. Within the XISS system indexes are used for speed up processing regular expression queries. A complex path expression is decomposed into several subsequent simple ones, which are then processed by one of the three join algorithms. The indexing scheme makes path query processing much faster than conventional algorithms. One unique feature of this approach is able to determine the ancestor-descendant relationship quickly and without traversal of the data tree. However, this scheme only deals with data that have tree-like structure. This means the data model of XML documents has to be a tree, i.e. the branches of the tree are independent, and there are no links between nodes of different branches.

The structure of the XML document can be considered as either syntactical or logical. In the first case a document is treated as a tree, and in the latter case the document can be viewed as a graph. However, only in some simple cases does an XML document have a logical tree structure where each of its elements has exactly one parent element. In general, the logical relationship between different elements within an XML document is a graph. Various models have been proposed with little difference (McHugh, 1999). If directly applied the numbering scheme proposed by Li and Moon (2001) on XML documents with graphs as their data models, requires significant additional query costs. For example, 0b shows a query that involves selecting node 2 and all of its reference nodes. If the numbering scheme proposed was used then 4 nodes need to be accessed. However, if the reference links can be modelled by the numbering scheme, then only two nodes need to be searched. Indeed, we have extended the numbering scheme to handle XML documents with various types of data models, including graphs and trees.

Extended XML Numbering Scheme

Securing graph structure describes the motivation for introducing an extended numbering scheme for XML documents and the details of the algorithms designed to maintain the indexes based on the extended numbering graph scheme.

Each XML document consists of a number of elements. Each element has its name, attributes, and content. Names of the elements are represented by tags, which are delimiters of the elements. The ancestor-descendant relationship is represented as nesting of the descendant element within the tags of the ancestor element. Due to the fact that the source data have a tree structure, where nodes of the tree correspond to the elements and labelled arcs stand for tags. Since such a tree represents the containment relationship between nodes of the document, it is referred to as a *containment tree*. An example of XML data and its interpretation as a tree are shown on 0a, 1b, and 1c.

The containment tree does not cover all semantic relationships between nodes but additional relations among elements can be defined through references or links. The name of the link describes its semantic role and the value of the link contains a path to the referenced node. The path sequence of names provides all ancestors of the node in the containment tree (W3C, 2001). Links are defined as the content of value of the attribute or the element. Depending on whether one needs to deal with the links as values or relations (arcs), the XML document can be treated either as a tree or as a graph (Papakonstantinou et al., 1995)0.

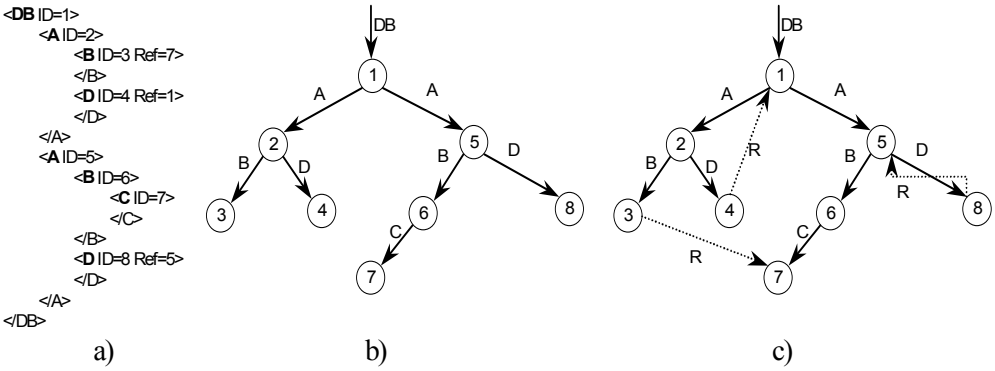


Figure 1 XML source data and its tree and graph representations

Tree Numbering Scheme

First, let us consider the trees that are involved in the representation of our approach. The idea of using numbering scheme to determine the ancestor-descendant relationship between any two nodes of a tree was firstly introduced by Dietz (1982)0. Each node of a tree is associated with a pair of numbers. These numbers are calculated during pre-order and post-order traversals of a tree. It is stated that given nodes x and y and their pre- and post-order traversal numbers ($n_{x\ pre}$, $n_{x\ post}$) and ($n_{y\ pre}$, $n_{y\ post}$) respectively, x is an ancestor of y (y is a descendant of x), iff $n_{x\ pre} < n_{y\ pre}$ and $n_{x\ post} > n_{y\ post}$. 0, shows an example of numbering of a tree. Such an approach to find the relationship between nodes is very useful for indexing of XML trees, since it avoids tree traversals during query execution. The pre- and post-order numbers of each node can be calculated and stored in a database. There is one disadvantage of this scheme: when a node is inserted, removed, or updated, the recalculation of all numbers will be required. To overcome this drawback, Dietz's numbering scheme was modified Li and Moon, to calculate and compare pairs of extended pre-order numbers and ranges of descendants (size of sub-tree) of any two nodes within a tree. "Extended" means that extra space are reserved for future possible insertions. Deletions do not require recalculation either, increasing only the available space within the parent node range. The proposition was: given nodes x and y and their extended pre-order and size numbers ($n_{x\ order}$, $n_{x\ size}$) and ($n_{y\ order}$, $n_{y\ size}$) respectively, x is an ancestor of y (y is a descendant of x), iff $n_{x\ order} < n_{y\ order}$ and $n_{x\ order} + n_{x\ size} \geq n_{y\ order} + n_{y\ size}$. The extended numbering scheme is applied to the same example shown on 0, b. For simplicity, each node on the example is assigned to the pair (n_{order}, $n_{order} + n_{size}$) instead of (n_{order}, n_{size}) – one can be obtained from another through simple arithmetical transformation. A bar originating from each node visually represents the range of the node.

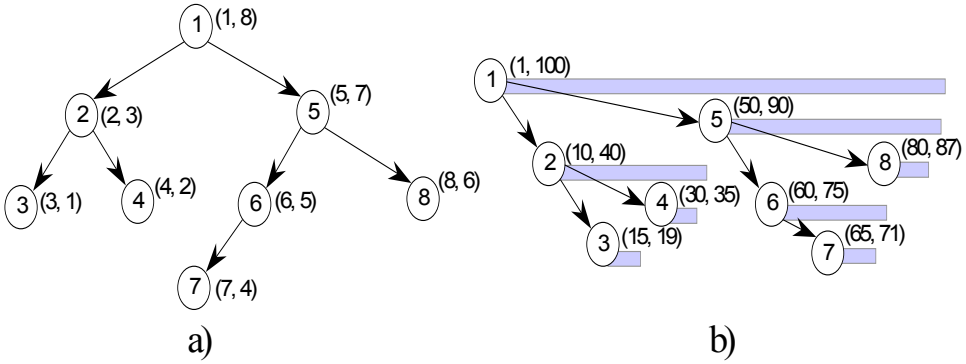


Figure 2 Numbering and extended numbering of the XML tree.

This numbering scheme was used to implement indexes in XISS project and had shown improved performance for processing queries.

Graph Numbering Scheme

We consider the same XML document as a logical graph (it is illustrated in 0, a). The graph is constructed based the numbered tree by interpreting links among its nodes. Such a numbering system could not be able to explicitly specify the ancestor-descendant relationship. For example, the link $o_4 \rightarrow o_1$ would create a cycle in the graph that makes the node o_1 an ancestor and a descendant of the node o_4 simultaneously.

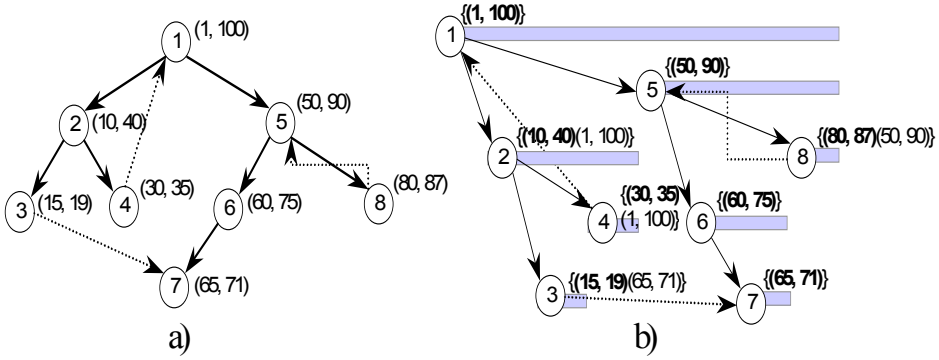


Figure 3 Numbering and extended numbering of the XML graph.

To resolve this problem, we propose a XML graph Numbering Scheme (XNS) that assigns a set of order-range pairs to a node instead of the single pair. The *numbering set* $\{(n^x_{order}, n^x_{size})\}$ of each node x of the graph consists of:

- A *primary numbering pair* $(n_{x\ order}, n_{x\ size})$ that is assigned to the node x during extended numbering of the XML containment tree. This pair is denoted as $(n^x_{x\ order}, n^x_{x\ size})$;
- A *linked numbering pair* is referred to as a primary numbering pair $(n_{y\ order}, n_{y\ size})$ of any node y that is linked by x directly or indirectly (through its descendants). Such a pair is denoted as $(n^x_{y\ order}, n^x_{y\ size})$.

The new linked pair $(n_{y\ order}, n_{y\ size})$ will be included into the numbering set only if it refers to the range not reachable through any other pair in the set, i.e. interval $[n_{y\ order}, n_{y\ order} + n_{y\ size}]$ is not contained in any interval $[n^x_{i\ order}, n^x_{i\ order} + n^x_{i\ size}]$ from the set $\{(n^x_{order}, n^x_{size})\}$. Note that the

numbering set of each node will never be empty and consists at least of one numbering pair, the primary numbering pair. The ancestor-descendant relationship between two nodes is determined as follows. Given nodes x and y and their numbering sets $\{(n^x_{order}, n^x_{size})\}$ and $\{(n^y_{order}, n^y_{size})\}$ respectively, x is an ancestor of y (y is a descendant of x), iff $\exists i$, $(n^x_{i\ order}, n^x_{i\ size}) \in \{(n^x_{order}, n^x_{size})\}$, that $n^x_{i\ order} < n^y_{y\ order}$ and $n^x_{i\ order} + n^x_{i\ size} \geq n^y_{y\ order} + n^y_{y\ size}$. In other words, there exist such a node i, which is referenced by x and the ancestor of the node y. An example of the graph numbering is given on 0, b. The primary pairs are shown in bold. Node o_8 has the linked pair (50, 90), corresponding to node o_5 and representing the link: $o_8 \rightarrow o_5$. Nodes o_2, o_4 have the linked pair (1, 100), corresponding to node o_1 and representing the reference $o_4 \rightarrow o_1$ (node o_2 as parent of o_4). Lastly, node o_3 has the linked pair (65, 71), corresponding to node o_7 and representing the reference $o_3 \rightarrow o_7$. Nodes o_2, o_4 are ancestors of the node o_3 but do not have the linked pair (65, 71) as they can reach node o_7 via their linked pair (1, 100). The root node o_1 has not any linked pairs: every node of the graph is reachable through its primary pair.

ALGORITHMS

This section describes algorithms designed to maintain index records for cross-references (links) of XML graph. Each record of the index has such a form $[o_x, o_y, (n_y, s_y), add\_info]$, where o_y, n_y, s_y are the identifier, the extended pre-order, and the size of the node y corresponding to the linked pair $(n^x_{y\ order}, n^x_{y\ size})$ of the node x. And *add_info* is the additional information stored in index, like IDs of the parent, first children, sibling etc. For simplicity, we omit *add_info* in the following discussion. The process of creating an index includes into two steps:

1. create the index of the XML graph nodes from the source data. This step will produce index records for primary pairs of each node of the graph;
2. create index records for linked pairs of each node of the graph. The pseudo-code of the algorithm is described in 2 below:

Algorithm 1 CREATE INDEX
Input: source XML data file
Output: index of the numbered XML tree

```
        //building tree index (records for primary pairs of each node)
1:      for each input element of the XML file do
        generate id, extended pre-order and size numbers o_i, n_i, s_i
        add index record [o_i, o_i, (n_i, s_i)]
        end for;

        //resolving links (records for linked pairs of each node)
2:      for each node o_i of the tree do
        for each link o_i → o_j do
                INSERT LINK o_i, o_j
                end for
        end for.
```

In the first step, the input source XML document is treated as a containment tree. The id o, the extended pre-order number n and the size s are generated for each node. Then the record $[o, o, (n, s)]$ is stored in a table. This record corresponds to the primary pair of the node. In the second step, the data is considered as a graph. Each node of the graph will be iteratively visited to resolve every reference (link) of the node. The appropriate records for the linked pairs of the node will be recorded. The Insert Link algorithm is designed to carry out the operations involved in this step, which is described in Figure 3 a. Each of the nodes in the graph has its primary pair. Let us start with adding link $o_3 \rightarrow o_7$. First, ancestors records of the node o_3 (including itself) are selected, including $A = \{[o_1, o_1, (1, 100)], [o_2, o_2, (10, 40)], [o_3, o_3, (15, 19)]\}$. Then select the set of linked pairs of the

node o_7: $D = \{[o_7, o_7, (65, 71)]\}$. For each ancestor $a \in A$, a record $[a, o_7, (65, 71)]$ is to be added in the index, if the node o_7 is not already a descendant of a. Thus the records $[o_2, o_7, (65, 71)]$ and $[o_3, o_7, (65, 71)]$ are added, and no record for the node o_1 will be added, since o_7 is already its descendant. Next, let we add link $o_4 \rightarrow o_1$. The ancestors records of the node o_4 (including itself) are selected, including $A = \{[o_1, o_1, (1, 100)], [o_2, o_2, (10, 40)], [o_2, o_7, (65, 71)]\}$. Then select the set of link pairs of the node o_1: $D = \{[o_1, o_1, (1, 100)]\}$. For each ancestor $a \in A$, a record $[a, o_1, (1, 100)]$ is to be added in the index, if the node o_1 is not already a descendant of a. The records $[o_2, o_1, (1, 100)]$ and $[o_4, o_1, (1, 100)]$ are added, and no duplicate record for the node o_1 need to be added. The record $[o_2, o_7, (65, 71)]$ will be removed since it is overlapped by the record $[o_2, o_1, (1, 100)]$. When we add the link $o_8 \rightarrow o_5$, the ancestors records of the node o_8 (including itself) are selected, including $A = \{[o_1, o_1, (1, 100)], [o_5, o_5, (50, 90)], [o_8, o_8, (80, 87)]\}$. Then select the set of link pairs of the node o_5: $D = \{[o_5, o_5, (50, 90)]\}$. For each ancestor $a \in A$, a record $[a, o_5, (50, 90)]$ is to be added in the index, if the node o_5 is not already a descendant of a. Thus, only the record $[o_8, o_5, (50, 90)]$ is added, and no records for nodes o_1 and o_5 need to bee added. The final result is shown on 0, b and the Algorithm 2 below shows the insert link.

Algorithm 2 INSERT LINK
Input: index, referencing node o_i and referenced node o_j
Output: index updated with records referencing node o_j

```
        //select ancestor records of the node o_i
        //note, that this selection includes o_i
1:      select index records for ancestors of the node o_i: A = {[o_p, o_r, (n_r, s_r)]}, such that o_p is the
parent of o_i;

        //select records of the node o_j
2:      select index records for of the node o_j: D = {[o_j, o_k, (n_k, s_k)]};

        //
3:      for each record [o_j, o_k, (n_k, s_k)] ∈ D do
                //
                for each record [o_p, o_r, (n_r, s_r)] ∈ A do
                        //
                        Ancestor ← o_p;
                        AddNode ← TRUE;

                        //
                while Ancestor = o_p do
                                //
                                if (o_r is parent of o_k) then AddNode ← TRUE;

                                //
                                if (o_k is parent of o_r) and
                                        (o_p ≠ o_r) then DELETE RECORD [o_p, o_r, (n_r, s_r)];
                        end while

                        //
                        if AddNode = TRUE then INSERT RECORD [o_p, o_k, (n_k, s_k)];

                end for
        end for
```

The algorithm for deletion the link $o_i \rightarrow o_j$ is straightforward: to delete all records of the form $[o_p, o_j, (n_j, s_j)]$, where o_p is the ancestor of o_i. This is to remove the records for links from parents of the node o_i to o_j and from o_i node itself to o_j. The pseudo-code of the algorithm is described in

Algorithm 3 DELETE LINK
Input: index, referencing node o_i and referenced node o_j
Output: index updated with records referencing node o_j

1: select index records for ancestors of the node o_i: $A = \{[o_p, o_j, (n_j, s_j)]\}$, such that o_p is the parent of o_i

//select records of the node o_j
2: **for each** record $[o_p, o_j, (n_j, s_j)] \in A$ **do**
//
DELETE RECORD $[o_p, o_j, (n_j, s_j)]$;
end for;

ANALYSIS

To assess the usability of this indexing method, we focused on the growth of index depending on the number of cross-reference links in XML documents. Because if the indexes growth dramatically and require enormous storage space, this method would not be applicable. We have chosen two real-world data sets (Digital Bibliography and Library Project (DBLP), and Internet Movie Database (IMDB)) to estimate the number of records in their indexes:

- DBLP contains computer science bibliography information. Each DBLP file represents a single publication. All the documents are grouped into classes of publication (book, conference paper, journal article, etc.).
- IMDB is a highly cyclic semistructured database with information about movies, actors, directors, producers, writers, etc. IMDB includes 49 files (>500Mb). We chose a subset of two files (movies, actors). The characteristics of the DBLP and IMDB data sets are summarised in **Error! Reference source not found.** below.

Data Set	Size (Mb)	Files	Elements	Links
DBLP	96.6	251,520	2,095,552	25,745
IMDB	88.1	2	2,221,423	1,624,311

Table 1 XML data sets

To estimate the index size S for a database, it is necessary to find numbers of records corresponding to primary $n_{primary}$ and linked n_{linked} numbering pairs of each node:

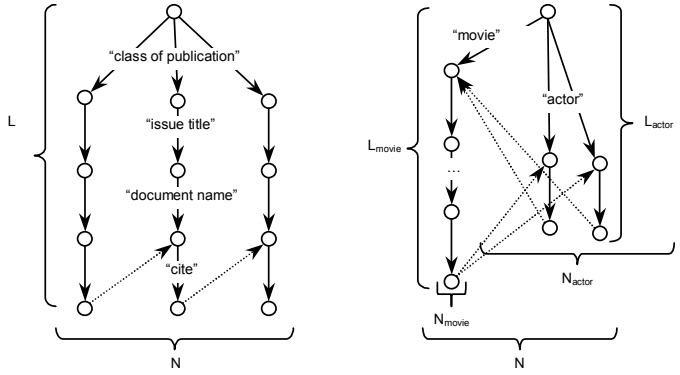
$$S = n_{primary} + n_{linked}.$$

The number of primary pairs $n_{primary}$ is equal to the number of elements in the database N. The number of linked pairs n_{linked} depends on the number of referencing nodes n_{ref} and how many ancestors precede these referencing nodes (according to the 0). The number of preceding ancestors of any node is equal to the path length of this node l_i. Consequently, the number of linked pairs n_{linked} equals: $n_{linked} = \Sigma\, l_i,\ i = [1, n_{ref}]$. If the path lengths of nodes are approximately equal to each other, the previous statement turns into the following:

$$n_{linked} = L \cdot n_{ref},$$

where L is the average path length of referencing nodes; n_{ref} is the number of referencing nodes. The number of referencing nodes n_{ref} depends on the topology of the graph, since every link increases the number of ancestors for the referenced node. For example, the number of ancestors of the node o_7 in 0: Numbering and extended numbering of the XML tree, a is equal to 3 (nodes o_1, o_5, o_6), while the number of ancestors of the same node o_7 in the 0: Numbering and extended numbering of the XML graph, a is equal to 7 (nodes o_1 to o_6).

First, we consider the topology of DBLP data. A DBLP document contains information about the type of publication, the title, the authors, and so on. It may also include a reference to other publication in the database. Each reference is given as an absolute path from the entry directory. The length of any path does not exceed 3 and has form «class of publication/issue title/document name». So the average path length of referencing nodes is $L_{DBLP} = 3$. The approximate topology of the DBLP dataset is shown on 0, a and b. It is assumed that there are no cross-references between any two documents, i.e. if book A has reference to book B, then book B can not have references to the book B.



a) b)

Figure 4 Approximate topology of DBLP data

The index size S for the DBLP database is:

$$S = N_{DBLP} + L_{DBLP} \cdot n_{BDLPref},$$

where N_{DBLP}, L_{DBLP}, $n_{BDLPref}$ are the number of elements, the average path length of referencing nodes and the number of referencing nodes in the DBLP data set. As a result of the number of elements, the increase of the index size of DBLP is a linear function and directly proportional to the number of references. The statistical results regarding the DBLP data set are summarised in Table 2 below:

Data Set	Number of primary pairs records, $n_{primary} = N_{DBLP}$	Number of references, n_{ref}	Number of linked pairs, n_{linked} ($L_{DBLP} = 3$)	Growth of index, %
DBLP	2,095,552 publications:251,520	1,000	3,000	0.1
		10,000	30,000	1.4
		100,000	300,000	14.3
		25,745 (present in DB)	77,235	3.7

Table 2 Estimation of Index Growth for DBLP data set

Since the average path length of referencing nodes in DBLP is low, the index size tends to grow slowly too.

Now consider the topology of IMDB data. There are several different types of IMDB documents: movies, actors, directors, etc. Each of the types has different structure and references to the documents of other types. The database contains many cross-references between documents. For example, "The Matrix" movie has links to the actors "Keanu Reeves" and "Laurence Fishburne". These actor documents have cross-reference to the "The Matrix" movie. We have selected the subset of the IMDB of two types (movie titles and actors). The approximate topology of the IMDB subset is shown on 0, b. Each type of documents has its average path length of referencing nodes L_{movie} and L_{actor}, and number of referencing nodes n_{movie} and n_{actor} respectively. If an average number of actors in each movie is denoted N_{actor}, then each of the N_{actor} actor nodes will have $(N_{actor} - 1)$ linked pairs to other actors and linked pairs to the movie. The movie node will have N_{actor} pairs for the actors. Thus, the number of linked pairs for one movie node and its ascendants is $L_{movie} \cdot N_{actor}$ and the number of linked pairs for one actor node and its ascendants is $L_{actor} \cdot ((N_{actor} - 1) + 1) = L_{actor} \cdot N_{actor}$. The total of linked pairs is:

$$n_{linked} = n_{movie} (L_{movie} \cdot N_{actor} + L_{actor} \cdot N^2_{actor}),$$

where n_{movie}, L_{movie} are the number of elements, the average path length of movie referencing nodes; N_{actor}, L_{actor} are the average number of actors and the average path length of actor referencing nodes. Even in the case of two types of documents, the index size grows non-linear. The statistical results regarding the DBLP data set are summarised Table 3 below:

Data Set	Number of primary pairs records, N_{IMDB}	Number of references, n_{movie}	Number of linked pairs, n_{linked} (N_{actor}= 20, L_{movie}= 5, L_{actor} = 3)	Growth of index, %
IMDB	2,221,423	1,000	1,215,000	54.7
	movies: 229,180	10,000	12,150,000	546.9
	actors: 367, 932	100,000	121,500,000	5469.4
		1,624,311 (present in DB)	1,973,537,865	88841.1

TABLE 3 ESTIMATION OF INDEX GROWTH FOR IMDB DATA SET

CONCLUSION

Our results showed that the indexing method introduced in this paper is efficient in cases where the number of cross-reference is relatively low as a result of securing the graph structure. However, in the case of circular references or where a large number of cross-referencing is involved, the number of index records will be increased at a non-linear rate. Indeed, the problems of Internet security cannot be ignored by companies as this would result in the loss of competitive advantage in the market place and what the future holds for Internet security technology cannot be predicted to the rate technology is advancing (Shoniregun, 2002).

REFERENCES

B. F. Cooper, N Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon (2001) A fast index for semistructured data. In **Proceedings of the 27th VLDB Conference***,* Roma, Italy

P. F. Dietz (1982) Maintaining order in a linked list. In **Proceedings of the 14th Annual ACM Symposium on Theory of Computing***,* San Francisco, California.

R. Goldman, J. Widom (1997) DataGuide: enabling query formulation and optimization *in* semistructured databases*. In* **VLDB**.

Q. Li, B. Moon (2001) Indexing and querying XML data for regular path expressions. In **Proceedings of the 27th VLDB Conference**, Roma, Italy.

O. Logvynovskiy , K. Lü (2003), Structural Sequence Join for XML Regular Path Expressions, **PREP'03**, Exeter, UK, April.

J. McHugh, J. Widom (1999) Query optimization for XML. In **Proceedings of the 27th VLDB Conference**.

Y. Papakonstantinou, H. Garcia-Molina, J. Widom (1995) Object exchange across heterogeneous information sources. In **Proceedings of the 11th International Conference on Data Engineering**.

Shoniregun C.A. (2002), The Future of Internet Security, ACM Ubiquity: Information Technology (IT) magazine and forum, Volume 3, Issue 37, Oct 29.

W3C (2001) XML Query Language (XQuery) 1.0. **W3C Recommendation**, December 20, http://www.w3.org/TR/xquery/ (Assessed date 21 may 2002).