SECURING XML QUERY PROCESSING STORAGE

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

Oleksandr A. Logvynovskiy
Business, Computing, and Information Management (BCIM),
London South Bank University
Borough Road, SE1 0AA, UK.

Kevin Lu
Department of Information Systems and Computing
Brunel University
Uxbridge UB8 3H, UK

ABSTRACT

The effective processing of semi-structured data queries is a preliminary part of data mining stage. XML queries employ regular path expressions to find structural patterns within XML documents. The operation of structural join is a crucial part of XML query processing. Existing approaches reduce complex join expressions to several binary structural joins. In this paper, we are proposing a new structural join algorithm called sequence join algorithm, for sequential regular path expressions in securing XML query processing storage. It exploits information about position of the elements in a product to skip generation of the redundant intermediate lists. This paper further discusses the algorithm that performs the merge of several input lists of nodes in one pass. We carried out comparative experiments, and the results prove that the algorithm is better than multiple binary joins algorithm for queries of both small and large cardinality.

Keyword: semi-structured data mining, data model, indexing XML data, sequence algorithm, graph numbering, and query processing.

INTRODUCTION

Each industry requires certain safeguards to protect its data while in transit. Bringing autonomic capabilities to storage systems would certainly be an improvement, but if computing systems that mine data in those storage repositories become next to impossible to manage, that partial automation will not yield much benefit. XML is now becoming a standard to represent and exchange semi-structured data over the Web (Xyleme 2001, Bray et al 2000). The problem of storing XML data in one or several tables is challenging, since the XML tree describe some irregular structure while tables are by definition regular. In a situation where the XML document has no schema, or when the schema changes frequently, it has a more dramatic impact on performance. The key idea behind these structures is called partial schema. The partial schema helped to build a concise graph representing paths of the data (Wang & Liu 2001). It also served as a guideline for building indexes and views or a starting point for structure-based document clustering based on approach proposed by Deutsch *e*t al (1999) that uses the XML instance to infer a relational schema. The idea is to find regularities that may exist in a given XML data instance, and to organise the storage base on those regularities (Deutsch et al 1999, Cooper et al 2001). The challenge in any storage schema is that it has to be flexible enough to accommodate data, and efficient as regular data storage. The number of accessible XML documents tends to grow as more and more business are storing and interchanging data between applications using XML as a common format.

SEMI-STRUCTURED DATA REPRESENTATION

We can represent semi-structured document as a graph or as a tree. The tree model reflects nesting of elements within XML file and treats reference elements (IDREF) as values. The graph model resolves semantics of reference elements and, thus, allows an element to have multiple parent elements. When semi-structured data is represent as a label, directed, and possibly cyclic graph, the vertices of such a graph correspond to objects that are either treated as containers for some other objects or associated with atomic values (such as text, multimedia content, etc.). Edges of the graph stand for containment relationship between nodes and have object types as labels. An example of such a model is the Object Exchange Model (OEM) (Papakonstantinou et al 1995), in this model, each XML element becomes an edge (labelled with the tag name) and directed towards an individual node. Each node corresponds to an XML element and has element's ID as its label. Each edge in the XML data graph has a label and a target, where the target is either a node representing a scalar data value (e.g. a string or integer) or a reference to an element node in the document via its ID. The edge belongs to a class, which can be one of sub-element, attribute, or IDREF (ID reference). Both IDREF and sub-element edges always direct towards element nodes in the graph; attribute edges. Every element node has precisely one incoming edge of sub-element type and any other incoming edges must be of IDREF type. If all IDREF edges of an XML data graph are converted into attribute edges (with destination value equal to the target node ID), the data graph can be mapped into a tree. Each element of the document forms a node in the tree labelled with the element type (tag name) and value. The edges of the tree stand for parent-child (containment) relationship between the elements. All sub-elements nested within an element appear in the tree as the child nodes directly connected with the edges to a parent node. The attribute in the elements represents in a nested sub-elements and form additional nodes in the tree, emanating from the associated parent nodes.

PATTERN SPECIFICATION LANGUAGE

A number of languages have been proposed for querying semi-structured and XML databases, which includes following XQuery, Lorel, XML-QL, and UnQL. A common characteristic of all existing language proposals is the existence of a pattern specification language e.g. Xpath, which built around path and sub-tree expressions. These expressions replace the traditional SQL FORM clause and enable selections based on value predicates as well as path navigation and branching through the XML data graph in order to reach relevant data elements. Path queries are popularised in the context of object-oriented databases, while the pattern specification language proposed for XML data are substantially more complex. In particular, the XPath language, Xquery, and XSLT, is the dominant W3C language proposals for XML querying and transformation, which allows branching of other regular path expression to enable queries navigation along the paths of data that uses label names, wild cards, value predicates and branching predications on the existence of specific product paths.

The key idea underlying the implementation of the existing join algorithms is the decomposition of the original query path expression into a set of simple (binary) path

expressions. Each binary expression produces an intermediate join result, which is used on the subsequent stage. The XISS system introduces three join algorithms: element-attribute (EA-join), element-element (EE-join), and Kleene-closure (KC-join). The element-attribute algorithm joins two intermediate results from sub-expressions, which are a list of elements and a list of attributes. The element-element algorithm joins two lists of elements. The principal difference between these algorithms is that the latter one checks ancestor-descendant relationship between each pair of the input lists while the former one tests parent-child relationship. The Kleene-closure algorithms iteratively uses element-element algorithm to compute closure of the expression. It repeatedly applies EE-join to the result from the previous stage of iteration. Both EA-join and EE-join algorithms have a loop over one input list nested into a loop over another list and, therefore, have time complexity $O(|E_1|\cdot|E_2|)$, which is quadratic in the size of the input lists. As KC-join depends upon EE-join, it has quadratic time complexity either.
However, structural join algorithms proposed by Al-Khalifa et al (2002) exploit the advantage of element numbering to decrease the time of processing (Al-Khalifa et al 2002). The tree-merge join algorithm is an extension of relational equality merge join performed on sorted inputs. The time complexity of the tree-merge join is non-quadratic $O(|E_1|+|E_2|)$, but may include multiple passes over the same input set of descendant nodes. To avoid this problem, the second of the proposed algorithms, stack-tree join algorithm, utilises stack of nodes and has time complexity $O((|E_1|+|E_2|)/B)$, where B is the blocking factor. However, semi-structured data imposes new challenges for parallel algorithms and requires new methods.

DATA MODEL

In representing semi-structured data, we use a graph data model labelled pseudo graph $G = \{V, A, L\}$, where $V = \{v_1, \ldots, v_n\}$ is a non-empty finite set of *vertices*, $A = \{(v_i, v_j)|v_i, v_j \in V\}$ is a finite set of ordered pairs of vertices called *arcs*, and $L = \{l_1, \ldots, l_k\}$ is a set of labels ascribed to vertices and/or arcs. Such definition of the database assumes that a graph can have loops and multiple arcs among its vertices. From the perspective of the database, a vertex of the graph is an *object* of the database and an arc is a *relation*. The data graph has an implicit order of its nodes obtained by the graph traversal. In order to make most of the graph numbering, we map original data graph into a directed cyclic graph. One of the important properties of the directed cyclic graph is that it has a cyclic ordering of nodes. We exploit this property to define a position of every node in the graph. Each object and relation within the database forms a *node* of the numbering graph. The edges of the graph stand for relationships between both of these elements, objects and relations. Relations that form cycles are reversed and relabelled. Position is an important characteristic of graph nodes and intensely used for indexing and querying semi-structured data. The *position* of the node n_i is denoted as (D_i, S_i, E_i, L_i), where D_i is the graph component identifier within the database; S_i, E_i are distinct graph ordering numbers of the node n_i (pre-order and post-order respectively), and L_i is the nesting depth of the node n_i within the graph. The ancestor-descendant relationship gives a graph node of $n_{i,}$ and position (D_i, S_i, E_i, L_i). The graph node n_j and its position (D_j, S_j, E_j, L_j), the node n_i is an ancestor of the node n_j (and node n_j is a descendant of the node n_i) if $D_i = D_j$ (both nodes belong to the same component), $S_i < S_j$ and $E_i > E_j$ (ancestor-descendant) or $S_i > S_j$ and $E_i < E_j$ (descendant-ancestor). Intermediate nodes n_x and their positions (D_x, S_x, E_x, L_x) that constitute path between the two nodes are select to meet the criteria:

$D_x = D_i = D_j$, $S_i < S_x < S_j$ and $E_i > E_x > E_j$ (node n_i is an ancestor of the node n_j); or

$D_x = D_i = D_j$, $S_i > S_x > S_j$ and $E_i < E_x < E_j$ (node n_i is a descendant of the node n_j).

The sibling relationship gives graph node n_i and its position (D_i, S_i, E_i, L_i). The graph node n_j and its position (D_j, S_j, E_j, L_j), the node n_i is a sibling of the node n_j (and node n_j is a sibling of the node n_i) if $D_i = D_j$ (both nodes belong to the same component), $S_i < S_j$ and $E_i < E_j$ (preceding) or $S_i > S_j$ and $E_i > E_j$ (following). The path between these two nodes is selected with respect to one of the common ancestors of nodes n_i and n_j, n_c and its position (D_c, S_c, E_c, L_c): $D_c = D_i = D_j$, $S_c < min(S_i, S_j)$ and $E_c > max(E_i, E_j)$. Then intermediate nodes n_x and their positions (D_x, S_x, E_x, L_x) are select to meet the criteria:

$D_x = D_c$, $S_i < S_x < S_c$ and $E_i > E_x > E_c$ or $S_c < S_x < S_j$ and $E_c > E_x > E_j$ (n_i precedes n_j); or

$D_x = D_c$, $S_j < S_x < S_c$ and $E_j > E_x > E_c$ or $S_c < S_x < S_i$ and $E_c > E_x > E_i$ (n_i follows n_j).

Finding a relationship between a pair of nodes is the core operation of the semi-structured data query processing. Graph numbering is the efficient way to determine it fast. Nevertheless, the scale of the real-world data challenges its wide application. As one of the solutions to overcome scale problem, we use graph layering as shown in Figure 1.

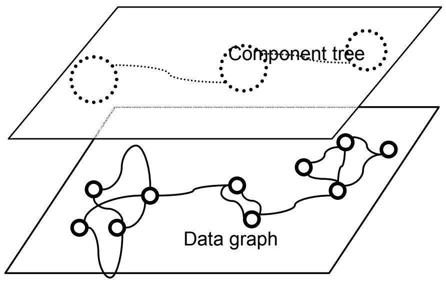

Figure 1. Layered data view

The original data graph is processed in order to determine its strong components. A strong component D of a graph G is a sub graph of every node in D and the removal of that node does not make D none connected. A node that ruins connectivity of the graph is a median. Nodes within a strong component are always connected. Nodes representing strong components of the graph along with medians form sets of trees. These trees are the next layer with its separate numbering. Every node of the original graph ascribed to the pair of positions, within the component tree. The procedure of finding whether a couple nodes are connected or not takes two steps: a) determine if their component are connected and if they are then b) determine their relationship within the component.

GRAPH NUMBERING AND QUERY PROCESSING

We have proposed the graph numbering using indexing for querying connections among data that exploits the concept of node position to merge several input lists in one pass. As it processes several binary structural relationships that form a sequence, we call it *sequence join* algorithm. The basic idea of the algorithm is to synchronously read input lists to find first match of the node intervals and put it into the result list. If the intervals in two adjacent lists do not match, based on the result of their comparison, the record of one of the lists will be deleted. The propagation of changes goes from the last list back to first. The depth of the recursion is equal to the number of input lists, which relates to XML data tree input. For regular path expression $a_1/a_2/\ldots/a_n$, both approaches require n selection operators resulting in lists of nodes $A_1, A_2, \ldots, A_n$ respectively. The execution plans for binary joins and sequence join are shown in Figure 2 below. We consider the total time to perform join operation as the sum of time needed to read input lists (σ) and time necessary to create output list (τ): $\sigma + \tau$.

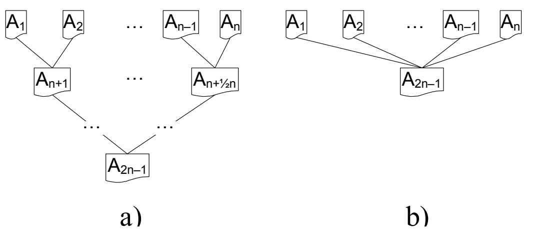

Figure 2. Execution plans of the binary and sequence joins

The task of matching complex query reduces the performance of join operation for each binary structural relationship in query expression. For n input lists of nodes, it causes creation of intermediate lists of nodes. The next step is to perform binary join operation over the intermediate lists. The latter will be applied until the result is the only one, in result list. Thus, the whole number of intermediate list (including the result list) is $n - 1$. In Figure 2 a) these lists are denoted as $A_{n+1}, \ldots, A_{2n-1}$. The total time necessary to perform multiple pair wise join is:

$$\left(\sum_{1}^{2n-2} \sigma_i + \sum_{n+1}^{2n-1} \tau_i \right)$$

The τ_i and σ_i is the time required to create and read list i respectively. The sequence join reads input lists $A_1, \ldots, A_n$ and project the result list A_{2n-1}, as shown in Figure 2 b). The total time necessary to perform sequence join is:

$$\left(\sum_{1}^{n} \sigma_i + \tau_{2n-1} \right)$$

The σ_i is time required to read input lists, and τ_{2n-1} is time needed to create the result table.

The parameters τ_i and σ_i depend on the capacity of the list i. If the number of nodes in each list is comparable then we can assume the times to create and read list are equal. The time differences between the two approaches in Figure 2 above is illustrated in Figure 3 below:

$$\left(\sum_{1}^{2n-2}\sigma_i+\sum_{n+1}^{2n-1}\tau_i\right)-\left(\sum_{1}^{n}\sigma_i+\tau_{2n-1}\right)=\sum_{n+1}^{2n-2}\sigma_i+\sum_{n+1}^{2n-2}\tau_i$$

Figure 3. Time differences

However, the parameters of τ_i and σ_i depend on the capacity of the list i. If the number of nodes in each list is comparable then we can assume the times to create and read list are equal: $\forall i$, $i = 1,\ldots,n$; $\sigma_i = \sigma$, $\tau_i = \tau$. Time cost functions of the algorithms is:

$(\sigma + \tau)(n - 2)$, (1a)

Where τ and σ is average time required to create an output and read an input list respectively, n is number of original input lists. The Figure 4 shows the time cost graphs of sequence join algorithm performance.

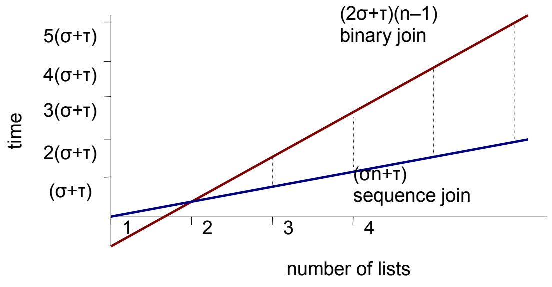

Figure 4. Time difference between the binary and sequence joins

The key idea of our proposed framework is to consider the original database as a graph:

- Both objects and relations of the database are represented as graph nodes – this provides a unified way for their arrangement, indexing and storage.
- The initial graph is direct and, generally, cyclic – we convert it into a directed cyclic graph by giving database relations status of nodes, then rearranging nodes so the graph is rooted.
- Every node of the graph ascribed with a couple of ordering numbers of the graph and post-order graph traversals analogous to a tree traversals.

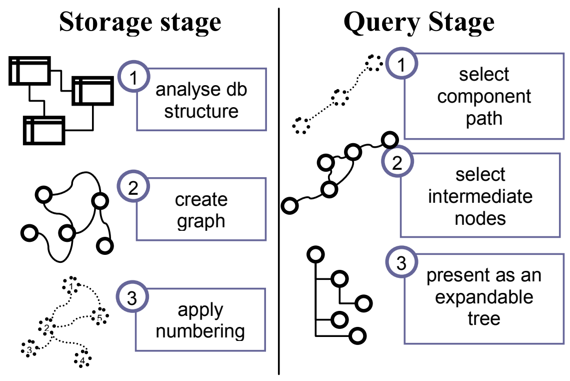

Figure 5. Framework of treating relational data as semi-structured

Figure 5 is performed has a pre-processing stage of storing new data or wrapping a legacy system. The query stage exploits graph numbering for fast selection of all intermediate nodes that constitute paths between nodes.

PROTOTYPE SYSTEM

We implemented a prototype system for storing, indexing, and querying connections among data. The system works as a wrapper for an existing relational database. The real world data set has been used as an input of the prototype. We used a subset of over 20,000 companies from a FAME (Financial Analysis Made Easy) database. The FAME database contains financial and statistical information about all companies in UK. We have limited our objects to company's directors, shareholders, postcodes and ownership data. The prototype screenshot shows the sample query of the semi-structured data to find connection between a person and a company (see Table 1 for sample queries).

Query	XQuery expression	Dataset	RPE length
Q1	/FMDataBase/FCCAmRadio/Address/City	HAM-RADIO	3
Q2	/PLAY/ACT/SPEECH/LINE	Shakespeare	4
Q3	/country/province/city/name	Mondial	4
Q4	/person/profile/interest/category	Xmark (100Mb)	4
Q5		Xmark (1Gb)	4

Table 1. Description and parameters of the test queries

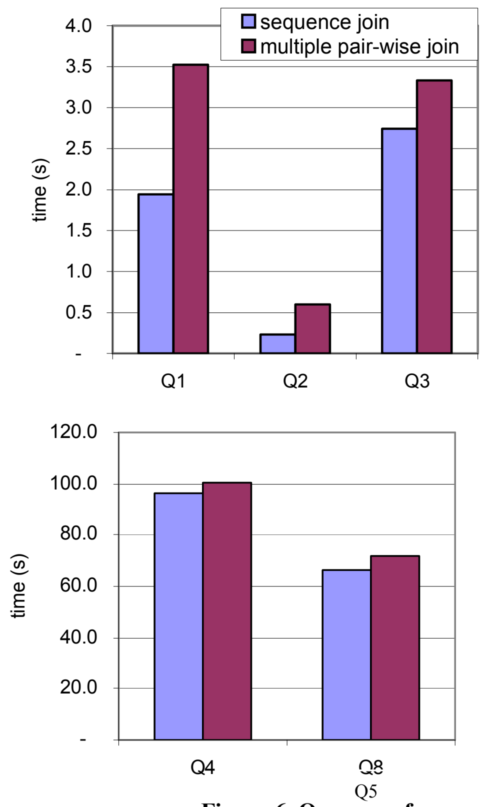

Figure 6. Query performance comparisons

The cases with the query path length of more than 2, demonstrates improved performance of the sequence algorithm against the pair-wise algorithm (see Figure 6). However, the selection of the result of all intermediate nodes that constitute paths between queries nodes are represented as a tree with expandable nodes.

CONCLUSION

Current approaches adopted the existing relational storages (and map semi-structured data into relational) or use native solutions. There exist hybrid solutions as well. Since any input query may initiate both mining processes and storage/retrieval operations, it is necessary to define appropriate criteria and algorithms for splitting/joining results obtained on each level. We have developed the sequence join algorithm for regular path expressions. In contrast to pair wise approach, the algorithm takes several lists of elements as an input and exploits the position of the element within XML document to compute faster structural relationships between elements. 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 (Shoniregun & Logvynovskiy2004). Therefore, the structural pattern are matched with available input lists at once but does not generate non-existent sub-results and hence eliminates creation of excessive intermediate data.

REFERENCES

Al-Khalifa, S. Jagadish, H.V. Koudas, N. Patel, J.M. Srivastava, D. and Wu, Y.,*(2002)* "Structural Joins: A Primitive For Efficient XML Query Pattern Matching", **In Proceedings of the IEEE International Conference on Database Engineering (ICDE).**

Bray, T., Paoli, J. Sperberg-McQueen, C.M., Maler, E., (2000) "Extensible Markup Language (XML) 1.0 (Second edition) ", W3c recommendation. Technical Report rec-xml-20001006, Available from http://www.w3.org/TR/REC-xml, (Access date: 29 October 2004)

Cooper, B. Sample, N. Franklin, M. Hjaltason, G. and Shadmon, M. (2001) "A Fast Index For Semistructured Data", In *Proceedings of VLDB'01*.

Deutsch, A. Fernandez, M. and Suciu, D. (1999) "Storing Semistructured Data With Stored", In **Proceedings of SIGMOD Conference**

Papakonstantinou, Y. Garcia-Molina, H. and Widom, J. *(1995)* "Object Exchange Across Heterogeneous Information Sources", In **Proceedings of the 11th International Conference on Data Engineering.**

Shoniregun, C. A. and Logvynovskiy, O. *(2004)* "Securing XML Documents", **The Australian Journal of Information Systems (AJIS***), September, Vol 11, pp 194-200.*

Wang, K., Liu, H.Q. (2001) "Mining is part of Association Patterns From Semistructured Data", In **Proceedings of the 9th IFIP 2.6 Working Conference on Database Semantics** *(DS-9)*, Hong Kong, April.

Xyleme, L. (2001) "A Dynamic Warehouse For Xml Data Of The Web", In **Bulletin of the Technical Committee on Data Engineering**, vol. 24, no. 2, June.

.