Semantic-based Construction of Content and Structure XML Index

Norah Saleh Alghamdi[1,2] **Wenny Rahayu**[3] **Eric Pardede**[4]

Department of Computer Science and Computer Engineering,
La Trobe University, Australia
Email: [1] nalghamdi@students.latrobe.edu.au
[2] n.alghamdi@tu.edu.sa
[3] w.rahayu@latrobe.edu.au
[4] e.pardede@latrobe.edu.au

Abstract

Content **A**nd **S**tructure (CAS) index for XML data is an important index type that has not been widely researched, even though its role is important especially in multi domain applications. Most existing researches in XML Queries Optimization focus on structure index alone. Few have utilized the rich semantic of XML data to support CAS index and querying. In this paper, we propose two indexes namely Structural index and Content index, whose construction utilizes XML data semantics and schema. These indexes contribute to a better CAS queries performance. The experiments prove that our method improves the performance of CAS queries by reducing the cost of CPU time and the total number of scanned elements compared to a standard method.

Keywords: Twig, Path, Index, Semantics, Objects, Value predicates, Content constraints, Structural predicates

1 Introduction

Most research on XML Queries is based on utilization of structural constraints for optimization (Liang et al. 2006, Haw et al. 2009, Ling et al. 2011). The use of structural constraints to improve XML queries performance is undeniable in report-based queries. On the other hand, for target-based queries, semantics constraints contribute to the query performance.

An example of a target-based query against Purchase Order XML data, in Fig. 2, is $Q1$ = *"find all customers who sent item by shipment to Melbourne, Victoria"*. This query is also called CAS since it requires a combination of content and structural constraints for its processing. A particular customer will be retrieved using structural constraints of the query and only a specific part of the customer's information will be finally retrieved utilizing the content constraints, which is also known as a value predicate. Without having value predicate, the query will list all customers information and generate a report which is out of the paper scope and target. Consequently, it is not practical to query without value predicates for transactional queries over large collection of data where each collection goes up to million or more of kilobytes. Therefore, an efficient method to index content or value is necessary to improve the performance of XML queries with value predicates.

To address queries with value predicates, content constraint alone is not enough. The knowledge of the whole path query is also required since the value predicate will not be standing alone without the existence of the whole path query.

Consider query $Q1$ above in XPath format *"/purchaseOrder/ShipTo [city= 'Melbourne' and state='VIC']/name/Fname"*.The significant characteristics are embedded within the query: (i) the underlying semantic structure between a set of interconnected nodes, (ii) the path connectivity between each of the nodes and (iii) the content constraints of the predicates. Therefore, it is important to take advantage from these query characteristics in optimizing the query performance. Because the nature of XML data and schema structure are rich in semantics, identifying and leveraging the semantic connectivity from the data and schema in process such query are beneficial in improving the query performance.

A naïve query processing by scanning the entire XML data in top-down or bottom-up fashion will cause significant performance degradation in most cases (Li et al. 2001). Indexing schemes have been developed in recent years to overcome this issue. However, most index schemes are capable at a certain phase of processing a query such as processing a simple path query without branches (Goldman et al. 1997) whereas others support a limited set of queries such as supporting only the structural queries without considering the value predicates (Haw et al. 2009, Liang et al. 2006). To the best of our knowledge, very few works have considered CAS queries in their indexing schemes. However, they do not utilize the semantics of XML data because they identified the XML nodes by global IDs which do not carry any semantic meaning (Li et al. 2001, Rizzolo et al. 2001, Zou et al. 2004, Monjurul et al. 2009, Chen et al. 2007). Therefore, exploiting the semantics of XML data to build an index scheme for CAS query processing is an ideal solution which has not been proposed in the past literatures yet. In this paper, we propose new indices which exploit the semantics of XML data and schema in their construction for efficient CAS query processing.

Our goal of this paper to achieve the following contributions:

- **Pruning the search space :** based on the semantic knowledge gained from XML Schema.

The first author would like to express her gratitude to the Taif University in Saudi Arabia, for supporting her with a PhD scholarship.

- **Building CAS index :** improving the query performance by loading merely the relevant portion of data during a query processing due to taking an advantage of the semantic nature of XML data in designing the value index.
- **Optimization :** producing the final results without the need to traverse the document leading to I/O cost safe.

The organization of this paper is as follows. Firstly in section 2, related works are discussed. In section 3, the preliminary knowledge for XML schema, data and query model is described. Section 4 describes our proposed index. Section 5 is the evaluation. Finally in section 6, the conclusion and future work are presented.

2 Related Work

Several indexing schemes have been proposed in order to improve XML path queries performance. However, despite the past efforts, the focus was in utilizing an index to assist in processing the structural part in twig queries effectively. However, these methods do not distinguish between the structural and content search. Leaf nodes with values and internal nodes without values in XML data have different characteristics, thus, processing the content in the same way as processing the structure will lead to expensive structural joins to search for contents. Add to this shortcoming, the semantic information of XML data has been ignored in the most previous studies for either value or non-value nodes. Therefore, this causes scanning unrelated portion of data that related to the query semantically.

Adjustable indices (Chung et al. 2002, Liang et al. 2006) group data nodes based on local similarity. However, since they keep track of the forward and backward paths, their size tends to be huge. Haw and Lee Haw et al. (2009) label each node of an XML document and then join them. However, their joining process becomes aggressive especially when the query has only ancestor-descendant relationships. ViST (Wang et al. 2003) and LCS-Trim (Tatikonda et al. 2007) transform both XML data and queries into sequences and then evaluate the queries based on sequence matching. The drawback of these methods is the occurrence of sequence matching.

XISS (Li et al. 2001) indexes XML data based on a pre-order numbering scheme to fastly determine ancestor-descendant relationship between elements. It consists of five components namely, element, attribute, structure, name indices and value table. XISS collects all distinct name strings in the name index implemented as B+-tree with "nid" as a name identifier. "nid" is used as a key for element index, attribute index and structure index. The main drawback of XISS is its nodes joining process, which can produce large intermediate results and in some cases may lead to query performance delay.

ToXin (Rizzolo et al. 2001) also collects values of XML data in a value index beside summarizing all forward and backward paths of XML graph in a path index. The value index contains of values and its corresponding nodes. The path index consists of the index tree corresponding to Dataguide (Goldman et al. 1997) and an instance function for each edge of the tree index. ToXin navigates down, navigates up and filters an XML query to produce a set of nodes that match a set of query nodes and relationships over value predicates. An obvious shortcoming is the lacking of support for range predicates since all values are treated as string. Since this approach uses Dataguide and uses edge approach to joining paths, it does not keep the hierarchy information to answer a complex twig queries.

Unlike ToXin, CTree (Zou et al. 2004) does not provide only path summaries at XML document level but also at the group level. It also provides details of child-parent links at element-level. In addition, CTree has multiple value indices per each data type of XML data including *(List, Number, DTime)*. All the value indices support a search *(value, gid, input parameter)* operation where gid indicates a certain group of CTree. By determining *gid*, irrelevant groups are eliminated in order to evaluate value predicates. Therefore, I/O cost is low. CTree can handle XML documents having only regular groups more efficiently. However, in the case of an XML document containing lots of irregular groups, the index space will rapidly rise, due to the need to element-level links for each element.

RootPath and DataPath index (Chen et al. 2007) can evaluate XML twig queries with value predicates to be tightly integrated with a relational database query processor. In RootPath index, the prefixes of the root-to-leaf paths are indexed. It is a concatenation of leaf value and the reverse of schema path, and it returns the complete node ID List. In contrast, DataPath index stores all sub-paths of root-to-leaf paths. In fact, the DataPath index is bigger than RootPath, due to the duplication of the schema paths and the node ID of its structure. The increase size of the index tends to rise accordingly with the increase of XML documents size. To overcome this shortcoming, (Chen et al. 2007) explored lossless and lossy compression techniques for reducing the index sizes.

To evaluate twig queries, TwigTable stores values in semantic-based relational tables whereas the internal structure of XML document is stored in inverted lists (Ling et al. 2011). In this approach, Structural join algorithm is used to maintain the inverted list while relational database processor maintains the tables. Semantic-based design of the tables brings performance advantages of TwigTable. On the other hand, the limitation of this approach is when a query does not have value predicates, no semantics will be applied and merely a structural join algorithm is performed.

3 Preliminary Knowledge

Schema and Data model: Practically, both XML schema and data are modelled as large, ordered *node-labelled tree T(N,E)* where each node *n*ϵN corresponds to an XML element and each edge between the nodes $(n_i, n_j)\epsilon E$ is used to identify the containment of node n_j under n_i in *T*. However, each leaf node ln_i of XML data contains a value denoted by $value(ln_i)$.

In Fig. 1, there is a purchase order schema which describes a purchase order generated by home products ordering and billing application (W3C 2004). In Fig. 2, there is the XML data tree corresponding to the schema in the Fig. 1. Each of them has elements connected by edges but the data has values in each leaf node.

Query Model: The focus of our work is on *CAS XML query* either a simple path or with branches. An XML query Q consists of nodes, labelled edges, and query predicate $Q(N_Q, E_Q, P_Q)$ where each node $q_i \epsilon N_Q$ represents a query tag that adheres to a set of an XML document's elements. A labelled

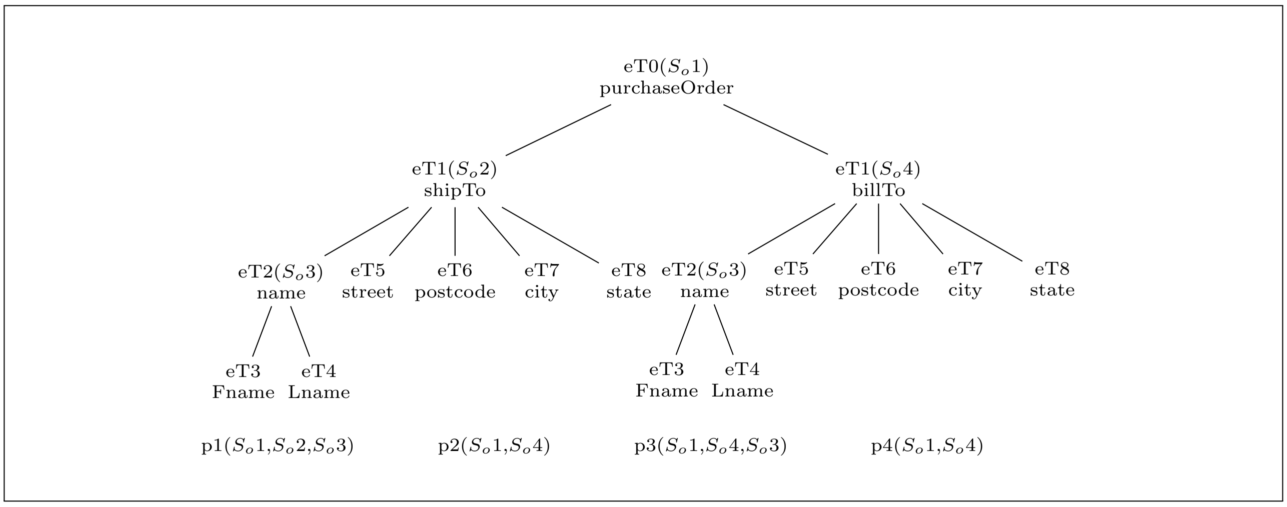


Figure 1: Purchase Order XML Schema

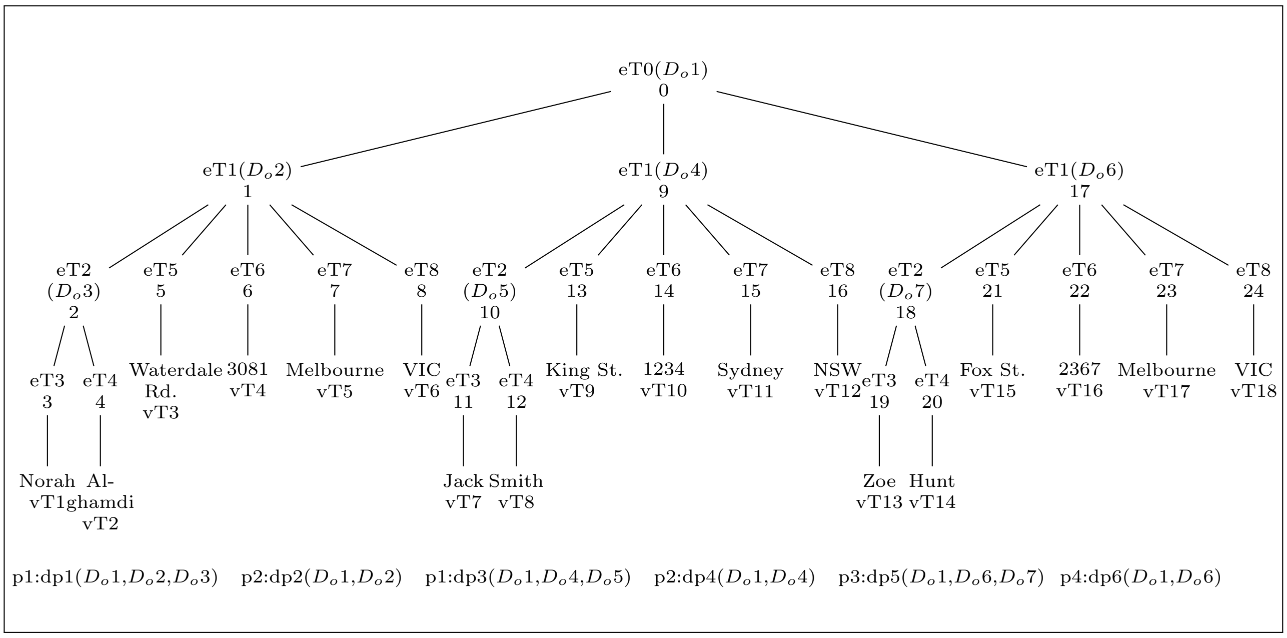


Figure 2: Purchase Order XML Data

edge between two nodes $(q_i, q_j)\epsilon E_Q$ indicates a structural constraint, which involves operators "/" and "//" denoting "PC" parent-child relationship and "AD" ancestor-descendant relationship respectively. A query predicate is held between brackets "[ ]" in the query Q including other structural constrains and a filter of content constraints. The filter of content constraints evaluates true based on the corresponding XML document nodes. A list of N-ary tuples is generated to produce a final result of matching Q to the XML document D, where N is the number of query tags and each tuple $(n_1, n_2, ..., n_k)$ contains the XML document nodes $n_1, n_2, ..., n_k$ which identify the matched results of Q in T.

Query Predicate: A query predicate P_Q is a combination of all or some of $(N_Q, E_Q)|(N_Q, E_Q, V_{PQ})|((N_Q, E_Q, V_{PQ}, P_Q)$ where each $q_i\epsilon N_Q$ is a query tag within the predicate brackets , each $e_i\epsilon E_Q$ is an edge between two query tags, and each $v_i\epsilon V_{PQ}$ is a value that can match a value of leaf node ln_i at the data model and P_Q is another predicate representing a branching point.

Consider a CAS query $Q2$ = "*//-shipTo[/state='VIC']/name*" where shipTo, state, name, /, // are structural constraints and 'VIC' is a content constraint. "*[/state='VIC']*" in $Q2$ is called a predicate which is a content constraint in this query and can be a combination of content and structural constraints.

4 Object-based Content and Structure XML Indexing

This section pays attention to utilizing the semantics of the structure and the content of XML data and schema during the index construction phase. Structural index is introduced first to maintain the structural constraints of XML queries. Thereafter, we represent the Content index which is proposed to improve the performance of querying constant values within XML data. Our methodology takes into account exploiting the semantic nature of XML data to improve the query performance. In order to achieve this goal, we adopt object-based XML data partitioning technique, called OXDP(Alghamdi et al. 2011), as pre-processing phase ahead of constructing the index. In particular, OXDP is semantic rich rules that can discover useful semantic information and identify objects within an XML schema. In this paper, we utilize such rules in determining XML document's objects and then partitioning the data based on the discovered objects. (refer to (Alghamdi et al. 2011) for more details). Indeed, the semantics term is used as everyday terminology by researchers across different concepts and application fields. However, to be precise, XML Semantics basically is envisioned in our work as an XML feature that enables us to identify XML data based on the meaning of their tags beside relationships between the tags. Such identification facilitates grouping and partitioning relevant data in order to provide semantically structured data.

Definition 1. (Object) *An object of an XML document is defined as a complex element type of XML schema associated with that document. In other words, an object is a non-leaf element that consists of simple or other complex elements.*

Definition 2. (Object-based Partition (Opart))*An Object-based Partition is a partition of XML schema and XML data that consists of a single object or multiple or nested objects.*

In Fig. 1 & Fig. 2, shipTo and its descendants are considered Opart1 in our example and billTo and its descendants are considered Opart2.

Afterwards, tokenising all distinct elements of XML schema as well as tokenising all distinct value inside XML data is taking the first place of constructing our indices. In a schema or a data, attributes and their associated values are treated as simple elements with values.

Definition 3. (Element Token (eT)) *An Element token is an identifier that encodes each distinct element's tag name of XML schema.*

Definition 4. (Value Token (vT)) *A value token is an identifier that indicates each distinct leaf element's value of XML data.*

In Fig. 1, each element of the schema is tokenised. For instance, the schema elements: "purchaseOrder", "shipTo" , "state" and "postcode" have "eT0" ,"eT1", "eT8" and "eT6" as their tokens respectively. Element Tokens represent all elements in the schema or the data. However, Value Tokens are created only from the data with values.. In Fig. 2, the values "Waterdale road", "3081" and "VIC" are tokenised to "'vT3", "vT4" and "vT6". Both eT and vT are implemented as integers to eliminate the computational overhead caused by the string comparisons but we use the symbolic eT and vT for a clear demonstration. The following subsections show the Structural indexing and the Content indexing.

4.1 Structural Indexing

Structural indices composing of Schema index and Data index are proposed mainly to cope with the structural part of XML queries in efficient way. These indices can evaluate arbitrary query structures including "/" or "//" as well as branching queries. The components of each index will be presented as follow:

Definition 5. (Schema index) *Schema index consists of element Tokens of the schema associated with a set of pairs consisting of Path of schema Object and ID of the Object-based partitions. Schema index can be represented as "(eT, (p, Opart))" where "eT" is the element token, "p" is Path of Schema Object and "Opart" is the partition ID where the element exists. The definitions of eT, p, and Opart are Definition. 3, 8 and 2 respectively.*

Definition 6. (Data index) *Data index consists of two indices. In the first index (Data index1), each Path of Schema Object is associated with all its corresponding of Path of Data Object. It can be represented as a pair as "(p, dp)" as following the Definitions 8 and 10. The second index (Data index2) consists of all Data Objects "D_o" grouped by each object partitions defined in 9.*

Definition 7. (Schema Object (S_o)) *a Schema Object is a set of tokens including an element token of a parent element tag, which is a complex element of XML schema, alongside the element tokens of its children tags. Consider S_o as the set of element tokens S_o (eT_{parent} , $eT_{child(1)}$, ...,$eT_{child(k)}$), where eT_{parent} is the element token of the parent node, $eT_{child(i)}$ is the element token of the parent associated children within the schema and k is the number of the parent's children.*

Definition 8. (Path of Schema Object (p)) *Path of Schema Object is a set of S_o located on the same path from the root to a leaf node of the XML schema.*

In Fig. 1, Schema Objects and Path of Schema Object have been added to the schema diagram. For instance, S_o2 is an Schema Object which its tokens is "eT1 eT2 eT5 eT6 eT7 eT8" of the elements "shipTo", "name", "street", "postcode", "city" and "state" respectively. It is important to highlight that in the Schema Object, we represent the object as a parent and its direct children tags and not its descendants. For instance, "name" has "Fname" and "Lname" as its children which have not been included in S_o2. In the same figure, the Path of Schema "p1" is a set of the Schema Objects S_o1, S_o2, S_o3 located in the same path from the root to the leaf.

Definition 9. (Data Object D_o)) *a Data Object is a pair of a set of element tokens associated with a set of those element positions inside XML data. Consider D_o (eT_{parent} , $eT_{child(1)}$, ..., $eT_{child(k)}$, Pos_{parent} , $Pos_{child(1)}$, ..., $Pos_{child(k)}$) where eTparent is the element token of the parent node, $eT_{child(i)}$ is the element token of the parent associated children and k is the number of the parent's children within XML data. The positions of XML data are generated during a depth-first traversal of the tree and sequentially assigning a number at each visit.*

Definition 10. (Path of Data Object (dp)) *a Path of Data Objects is a set of D_o located on the same path from the root to a leaf node of the XML data.*

In Fig. 2, Data Objects and Path of Data Objects have been add to the XML data tree. For instance, D_o2 is an Data Object which its tokens is "eT1 eT2 eT5 eT6 eT7 eT8" with their position "1 2 5 6 7 8". In the same figure, "p1" consisting of "S_o1, S_o2, S_o3" in Schema index is corresponding to two Paths of Data Objects "dp1" and "dp3" containing "Do1 Do2 Do3" and "D_o1 D_o4 D_o5" as a set of Data Objects respectively.

4.2 Content Indexing

To index the content of XML data, Value index is proposed. Firstly, the values are tokenised and stored

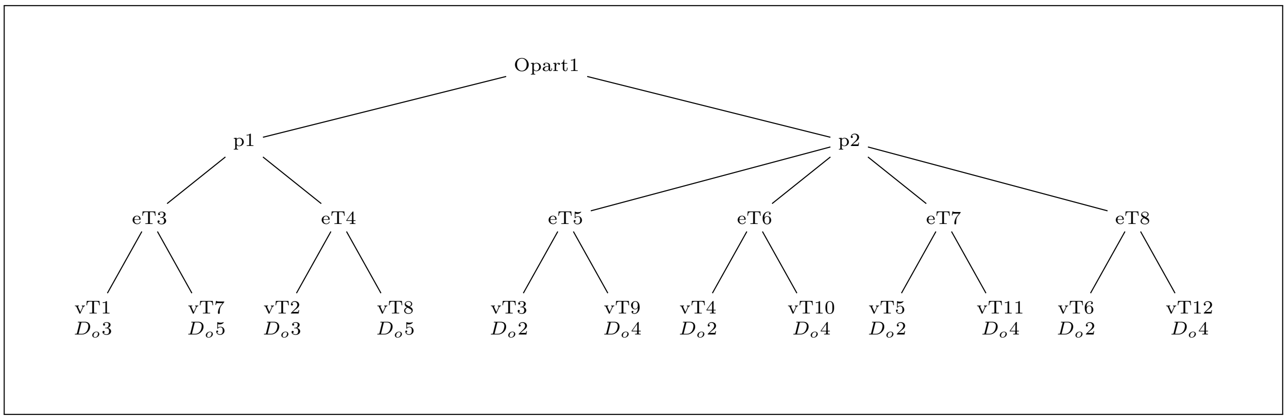


Figure 3: Value Index

according to their schema design. Secondly, the index stores all the value tokens with their corresponding Data Objects according to their data context.

The Value index is built from the Schema index, Data index and XML data. It keeps the semantic connectivity between the nodes of XML data to produce an efficient performance for a query with value predicates. It consists of all the object partition identifiers associated with a set of Paths of Schema Object. For each path, there are all the element tokens of only the leaf nodes. The value tokens of the corresponding element token are associated with their Data Objects.

Definition 11. (Value index) *Consider a Value index as VI = $Opart_1$,..., $Opart_k$ where $Opart_i$ is an object partition identifier and k is the number of object partitions. Consider for each object partition identifier of VI as $Opart_i$ = $Leaf(p_1)$,..., $Leaf(p_m)$ where each $Leaf(p_j)$ is a Path of Schema Object exists in $Opart_i$ and m is the number of Path of Schema Object. Let $Leaf(p_j)$= eT_1,..., eT_n is a Path of Schema Object consisting of a set of tokens of leaf elements in the XML data and n is the total number of element tokens. For each eT_i, there is a set of value token associated with its corresponding Data Objects eT_i= $<vT_1,\{D_{o1},...,D_{on}\}>$,... , $<vT_y ,\{D_{o1},...,D_{on}\}>$ where y is the number of value tokens and n is the number of Data objects per each value tokens.*

Figure 3 depicts the Value index in only "Opart1", the rest of the partitions will have similar representation. "Opart1" has "p1" and "p2" as its Path of Schema Object where each of them has the tokens of the leaf elements "eT3, eT4" and "eT5, eT6, eT7, eT8" respectively. It can be seen that "eT2" in "p2" was not considered since it is an element token of non-leaf node. The last level of this index is the association of the value tokens and its associated Data Objects <vT1,D_{o3}> and <vT7, D_{o5}> depicted in the same figure.

The redundancy of the data within an XML document increases the index size in most of the previous works. In our index, this issue have been taken into consideration as shown by Remark 1.

Remark 1. *There is a single value token for all matched values located in the same object partition, same Path of Schema Object and having the same element tokens.*

From Remark 1, we can see that the memory can be saved thus, the searching time will be reduced as well. Considering this remark will be more practical for the data that often has a redundancy. For instance, in a small scale, let say the purchaseOrder XML data has 10 shippers living in the same shipTo address, it is more precise to ignore the redundant data of the address and record the address only once. In the large scale, this redundancy will affect the performance of a query processed over the index negatively. It is important to highlight that Data Objects associated with each value token will assign the position of that value's parent node within the data as the Definition 9 . This feature will improve the efficiency of processing the query by trimming the search space in the Data index later.

4.3 Discussion

With both Schema index and Data index, either single-path queries and branching queries can be answered. For instance, consider $Q3$ ="*/purchaseOrder/shipTo [/street][/state]*", the elements tokens of the query are "eT0" ,"eT1", "eT5" and "eT8". From the Schema index," eT0" is associated with (p1, Opart1), (p2, Opart1), (p3, Opart2), (p4, Opart2), (p5, Opart3), eT1 is associated with (p1, Opart1), (p2, Opart1) and eT5 and eT8 are associated with (p2, Opart1). To trim the search space, we do an intersection based on Opart between "eT0" and "eT1", then between the intersected result and each child of "eT1" separately i.e. "eT5" and "eT8". The final result will be (p2, Opart1). From the Data index, "dp2" and "dp4" are retrieved based on the "p2" of the Schema index. Since "dp2" and "dp4" consist of a set of Data objects, the position of the query elements will be retrieved from the Data Objects, which are located in Opart1, by matching the element tokens of the query with the element tokens of the Data objects. The final results of the query would be "0, 1, 5, 8" and "0, 9, 13, 16".

The Value index in conjunction with Structural indices can be used to evaluate arbitrary queries with different predicates such as a value predicate, single path ended with value predicates or a branched path ended with value predicate. The functionality of Value index will be discussed using this part "shipTo[/street= 'Waterdale Rd' " of the query *Q3*. Since the Value index groups values of an XML document within objects. By apply this semantic-based technique, the search will be trimmed semantically based on each objects. In the given query, the Schema index of "street" is eT8 associated with (p2, Opart1). This means that only Opart1 is required. In addition, the index adds the Path of Schema Objects as identifiers that assist in decreasing the search space of the values. In our example, the search space will be trimmed to those that have "p2" within the partition "Opart1". Another advantage is that instead of preceding an aggressive string search looking for matching values to the query condition, element tokens will eliminate irrelevant values. Thus, the condition 'Waterdale Rd' will be mapped to its value token and then the token will be looked at from eT8 without the need to scan all the value tokens within the path "p2". The output of this index is the value token, which is "vT3" in the query, leading us to the right Data Object which is D_o2. The interest of finding the Data Objects, i.e."D_o2", is that only related Data Objects will visit in the Data index to produce the final results.

The design of the proposed indices has three features to facilitate the evaluation of twig queries in an optimum execution time. The indices are able to: (i) preserve the details of parent-children elements through the objects, (ii) preserve the details of all objects located in each path of the schema and data as in Path of Schema Object and Path of Data Object, and (ii) partition and keep links between interconnected data based on object based semantics as stated in Definition 2.

4.4 Algorithms

Our algorithm "ProcessQuery" is a recursive function building decomposing a branched path into multiple single paths. It applies the intersection based on the objects from the Schema index of all query nodes "qNode" located on the same query path. This process will end up with intersected Schema index i_SI among all the query nodes within the path to help us to use the information of the Path of Schema Objects "p" and the object "OPart" where the search will be done on.

Then, it evaluates the path ending with a value predicate using the function *"EvaluateContent"* and the path without a value predicate is evaluated by *" EvaluateStructure"*. Both EvaluateContent and EvaluateStructure are used in finding the XML node positions within the data. They are similar in their independent evaluation of each other, i.e. the whole path can be processed completely by only one of them without the need to the other. The only difference between them is that EvaluateContent utilizes the Value index for finding the candidate Data Objects which hold the condition of the value predicate before the proceeding the structural search. Thus, we can say that EvaluateStructure can do only the structural search whereas EvaluateContent can do both structural and content search. The main advantage of EvaluateContent is its capability to trim the search space of scanned elements. The details of EvaluateContent will be shown later. After evaluating the content and structure of the query and getting the XML nodes positions, the result will merged through MergeResult which keep the structural order of the node and produce fine results. The function EvaluateContent can do two main functionalities. The first is the content search which starts from line 1 and then embeds the structural search from line 8. The content search uses Value index to retrieve only participated Data Objects by filtering the value predicate using the information coming from the Schema index as "p" and "OPart" . After that, the structural search will be done on those XML nodes that exist within the participated Data Objects. At line 8, it goes through the related portion of the first Data index that matches "p" to check the last D_o of the

```
   Input: qNode, c_SI "current Schema index",
          path, depth
   Output: Query nodes position within the data
 1 if ¬qNode.Children then
 2     path.Add(qNode);
 3     i_SI ← Intersect(c_SI,SchemaIndex[qNode]);
       foreach p of i_SI do
 4         if qNode.ValuePredicate then
 5             xNums ←
               EvaluateContent(p,path,Opart)
 6         else
 7             xNums ←
               EvaluateStructure(p,path,Opart);
 8         end
 9         res.Add(xNums);
10     end
11     return res;
12 end
13 i_SI ← Intersect(c_SI, SchemaIndex[qNode]);
14 firstOccurrence ← true;
15 path.Add(qNode);
16 foreach c in qNode.Children do
17     temp ← ProcessQuery(c, i_SI, path,
       depth+1); if firstOccurrence then
18         result ← temp; firstOccurrence ← false;
19     else
20         result ← MergeResult(result, temp,
           depth);
21     end
22 end
23 return result;
```

Algorithm 1: ProcessQuery

dp if it is matched with D_o coming from Value index as line 10. The search space is narrowed to those dp that end with thatD_o of Value index. Then, at line 21, each D_o of Data index 2 will be visited with regarding to D_o of Data index1 as in line 22. As its know that each D_o consists of the tokens of the nodes within that D_o associated with their positions. Lines 24-25, if a query node token is matched with a node token of D_o, its position is added to the temporal result collection. Once the leaf node of the query is reached as at line 38, the temporal result will be added to the list of the final results.

5 Experiments

5.1 Environment set up

In order to study the improvement in CAS query processing by our proposed indices, a series of experiments was carried out. Our Value index performance is compared with the standard value index with the focus on the value predicates. The structural part of the query in both methods is the same. We use our Structure index including Schema index and Data index to process the structural part of the query and the content part processing is done by both our Value index and the standard value index.

A prototype system was implemented using C#.Net. All XML indices in this paper were loaded into RAM before running the queries, thus IO cost of reading the index data are not required. All the experiments were conducted on an Intel Core 3.2 GHz PC with 6.00 G RAM running Windows 7.

```
   Input: p:"Path Of Schema Objcet", quP: "query path", Opart: "Object partitions"
   Output: XML nodes positions
 1 foreach viObject in ValueIndex[OPart] do
 2     viPath = viObject[p];
 3     viToken = viPath[quP[quP.Count-1].Token];
 4     if !viToken[quP[quP.Count-1].vPred] then continue;
 5     $q_v$= quP[quP.Count-1].vPred;
 6     for $vT_i$ ϵ viToken[$q_v$] do
 7         d = viToken[$q_v$][i];
 8         PathsOfDo = DataIndex1[p];
 9         foreach dp in PathsOfDo do
10             if dp[dp.Count-1] != d then continue;
11             TmpR = new List (); k = 0;
12             for $D_{o_j}$ ϵ dp do
13                 (find matched element tokens continue from line 20 to 47);
14             end
15             if ($R_t$.Count=quP.Count) Result.Add($R_t$);
16         end
17     end
18 end
19 return Result;
20 object=DataIndex2[viObject];
21 foreach $d_o$ϵ object[$D_{o_j}$] do
22     for $eT_i$ ϵ $d_o$.Tokens & k < quP.Count do
23         if $eT_i$=quP[k].Token then
24             $R_t$.Add($d_o$.Pos[$eT_i$]);
25             if k > 0 then
26                 if quP[k].Tag[0] != '/' then
27                     $y_p$=nDepth[$R_t$[$R_t$.Count-1]];
28                     $y_c$=nDepth[$R_t$[$R_t$.Count-2]];
29                     if $y_p$ != $y_c$+1 then
30                         $R_t$.Remove($R_t$.Count-1);
31                         continue;
32                     end
33                 end
34             end
35             k++;
36             if k=quP.Count&quP[k-1].leaf then
37                 if $R_t$.Count=quP.Count then
38                     $R$.Add(new ($R_t$));
39                 end
40                 $R_t$.RemoveAt($R_t$.Count-1);
41                 k- -;
42             end
43             if k=quP.Count || quP[k].leaf then break;
44         end
45         if !quP[k].leaf then break;
46     end
47 end
```
Algorithm 2: EvaluateContent

Table 1: Different characteristics of each dataset

Parameter	DBLP	Auction	SigmodRecords
#Nodes	3332131	157	11527
Depth max.	6	5	6
#Fan-out	22	5	4
#Distinct Elements	36	32	11
Opart	8	2	2

5.2 Standard Value Indexing

In most past research, the standard indexing method for values when value predicates exist in the CAS queries is to index each value with its node position id. When performing joins, a small amount of node ids will be returned for further joins. We compare our proposed value index with the standard value index.

5.3 Experiment Datasets

DBLP, Auction Data, and SigmodRecords were used in our experiments and were obtained from XML repository of University of Washington University of Washington XML Repository (2002). Different characteristics of each dataset is shown in Table 1.

5.4 Experiment Metrics

To evaluate the performance of our proposed algorithms, two metrics were used. The first metric is obtaining CPU cost by calculating the average execution times of a query. Secondly, the total number of scanned elements is measured during a joining process. This metric will provide a good reflection about the ability of our algorithms to trim search space and to skip portion of the data.

5.5 Experiment Criteria

Since the focus of this paper is the performance of CAS queries, the evaluation considers the use of both structural and content index on the queries. To examine the structural part, we vary the type of relationships Parent-Child (PC) or hybrid of PC and Ancestor-Descendant (AD), and the number of branches. To evaluate the content part, we consider value predicates composing of numeric or string. Added to these criteria, we included simple paths and branch paths with values in the predicates.

5.6 Experiment Queries

Table 2 presents the evaluation queries. Each query is coded “QXN”, where ‘X’ represents ‘S’ (SigmodRecords), ‘D’ (DBLP), or ‘A’ (Auction Data), and ‘N’ is the query number within the respective dataset. The queries were selected to cover most combinations of the evaluation criteria, thus, the sensitivity of the query performance can be indicated to each criteria. QD9 and QD11 are simple path queries whereas the rest are branch queries. QD6 and QS5 contain only PC relationship, while the rest contain hybrid edges of PC and AD. We have a variety of branches number in the branched queries. For example, QS1 and QS2 have two branches whilst QD3, QD4, and QD5 have 3,4,5 branches respectively. The type of value predicates is also different among the queries, QA9 and QS5 have path-value predicates while QA10, QS4, and QS5 have path-branch-value predicates and the rest are merely value predicates.

Table 2: Experiment of Queries

QXN	Query pattern
QA1	//auction_info[/current_bid=“ $620.00”]/time_left
QA2	/root//auction_info[/location=“ LOS ANGELES, CA”]/high_bidder/bidder_name
QA3	/root/listing[//location=“ LOS ANGELES, CA”]/auction_info/time_left
QA4	//auction_info[/current_bid=“ $620.00][/num_items=“1”]/time_left
QA5	//auction_info[/current_bid=“$610.00”][/num_items=“1”][/started_at=“ $100.00”]/time_left
QA6	//auction_info[/current_bid=“$610.00”][/num_items=“1”][/started_at=“$100.00”] [/num_bids=“16”]/time_left
QA7	//auction_info[/current_bid=“$610.00][/num_items=“1”][/started_at=“$100.00”] [/num_bids=“16”][/location=“Allentown, PA 18109 ”]/time_left
QA8	//listing[/seller_info/seller_name=“ cubsfantony”]/auction_info/current_bid
QA9	//listing[/auction_info[/current_bid=“$620.00”][/num_items=“1”]] /bid_history/quantity
QS1	//issue[/volume=“11”]/number
QS2	//article[/title=“Architecture of Future Data Base Systems.”]//authors
QS3	/SigmodRecord[/issue[/volume=“11”][/number=“1”]/articles//title]
QS4	/SigmodRecord/issue//article[/initPage=“30”][/endPage=“44”]/title
QS5	/SigmodRecord/issue[/articles/article[/endPage=“44”][/initPage=“30”]]/volume
QD1	//article[/author=“Frank Manola”]/title
QD2	//article[/editor=“Paul R. McJones”]/title
QD3	//article[/editor=“Paul R. McJones”][/volume=“SRC1997-018”]/title
QD4	//article[/editor=“Paul R. McJones”][/journal=“Digital System Research Center Report”]/year
QD5	//article[/editor=“Paul R. McJones”][/journal=“Digital System Research Center Report”][/volume=“SRC1997-018”]/year
QD6	/dblp/article[/author=“Tor Helleseth”]/year
QD7	/dblp/inproceedings[/author=“Tor Helleseth”]/title//sub
QD8	/dblp/inproceedings/title[/i=“C”]/sub
QD9	/dblp/inproceedings/title[/sub=“INF”]
QD10	/dblp/inproceedings/title[/sub=“INF”]//sub
QD11	/dblp/inproceedings//i[/sub=“n, n”]

5.7 Performance Evaluation

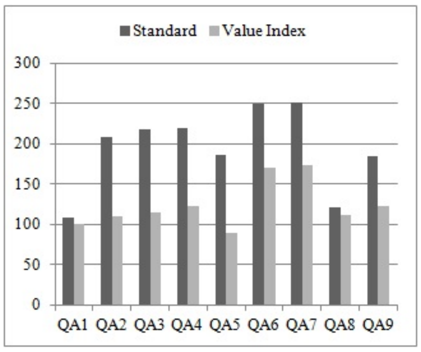
In this section, the efficiency of the Value index which is based on the objects has been studied. As mentioned above, the system supports search by content and structure. To achieve this goal, our index provides mechanisms to process the content and structure efficiently. Structure and Content Indexes are combined to answer regular path queries with predicates over values.

We rely on our indices in finding the value predicates before finding and matching the node position. The rationale is that content search normally results in high selectivity. By performing content search first, we can reduce the complexity of structural joins. A content search based on the specified value predicates comparison works as a filter prior to the structural search.

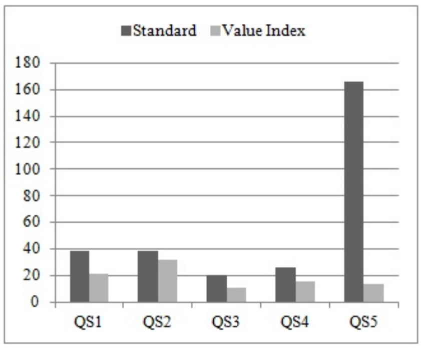
5.7.1 CPU Time.

We compare the time performance of our object-based value index with the standard value index. In Figure 4(a), the experiments run on Auction data set. The queries represent a combination of different criteria as mentioned in Section 5.6.

Our index is 2 to 4 orders of magnitude more efficient than the standard one in all queries. For instance, while our index takes about 0.1735 milliseconds to retrieve one answer of QA7, the standard index, when querying data of the same size, takes almost 0.2518 milliseconds. The standard index performs well since it uses our structural index to search the structural part of the query. However, our method performs better by combining the strength of object-based structural index with the strength of object-based Value index. The objects in our Value index carry semantic meaning and in each value is stored based on their paths and tokens within an object to provide fast access to right values that match to the

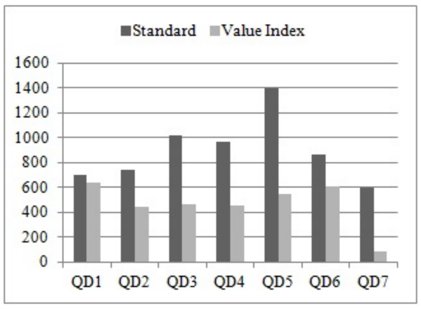


(a) Auction data (Time is in microseconds)

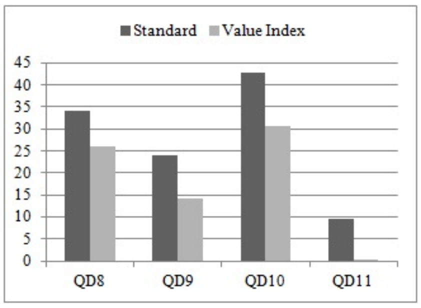


(b) SigmodRecords data (Time is in milliseconds)

Figure 4: The elapsed time for datasets



(a)



(b)

Figure 5: The elapsed time for DBLP data set (in milliseconds)

value predicates. This is in contrast to using the standard value which does not carry any semantic meaning leading to consume the search time for finding the right matched values.

Figures 5(a) and 5(b), show the execution time of the queries evaluated over DBLP data set. Our experiments reveal that our Value index outperforms the standard value index. For example, QD6 retrieves 15 results from data of 3332131 node. Our index needs about 600 milliseconds to produce the results while the standard requires around 900 milliseconds. Since the standard value index is node based index, it implies that increasing the total number of nodes increases the size of the data to scan and check. This is because it does not have a specific technique assists in skipping irrelevant portion of data which exists in ours. Our Value index trims the search space based on the objects. For instance, in QD6, our method needs to access only the part of the data that related to "article node. However, the standard method searches based on the node and needs to access many nodes which do not participate in the final results.

The results of Figure 4(b) support the earlier experiment outcomes. Our index took less time to evaluate the queries over SigmodRecords dataset. We can notice that QS5 has a significant performance because all the relationships between the query nodes are P-C while other queries are hybrid of P-C and A-D. We can observed a big difference in the performance because of the type of relationships. Our method gains more benefits in improving the query performance from P-C relationship than the standard does.

5.7.2 The total number of scanned elements.

The main purpose of this experiment is to indicate the capability of our index to avoid scanning irrelevant portion of the data.

Figure 6 is the total number of visited elements during the evaluation of Auction data. The total number in our index is reduced by 68% compared to the standard index. As the same outcome, Figures. 7(a) and 7(b) shows that the total number of visited elements are decreased by 59.6% and 77.7% for DBLP and SigmodRecords respectively. This is an evidence of the efficiency of exploiting the semantics of XML data in constructing the Value index. Since all the query nodes has a semantic connectivity between them that has a similar representation with the data, our Value index utilizes this significant which effect in the reduction the search space.

The total number of scanned elements is also affected by the type of relationships. In QS5, the average reduction is 98.5% of the total number scanned by each method. This leads us to a conclusion that the high selectivity causing by P-C relationship reduce the number of elements to check in order to produce the result.

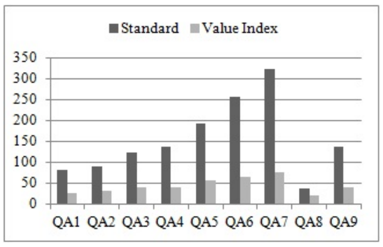
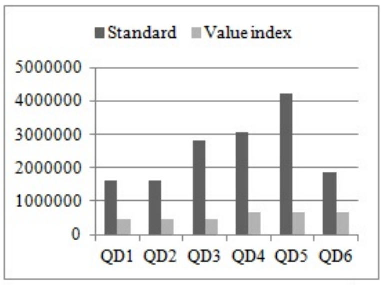
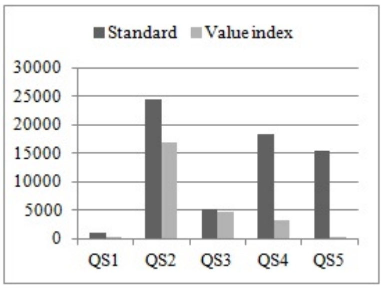


Figure 6: The total number of scanned elements for Auction data set



(a) DBLP dataset



(b) SigmodRecords dataset

Figure 7: The total number of scanned elements

5.7.3 Changing the number of branches.

We select QD5 from DBLP, with four branches and then varying the number of branches from 2 to 4 as in Figure 8. The CPU time to process the queries is shown in Figure 9(a) whereas the total number of checked elements are shown in Figure 9(b). It can be seen that the CPU cost of both methods increases as the number of branches in the queries increases. However, the cost of standard value index is much more than the cost of the Value index. This is because that standard index needs to scan elements more than ours as shown in Figure 9(b).

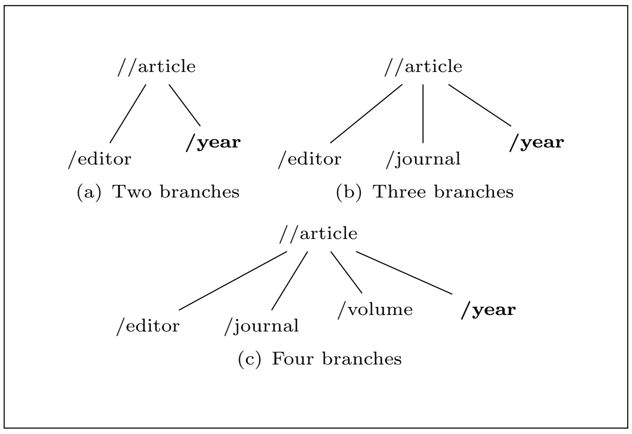
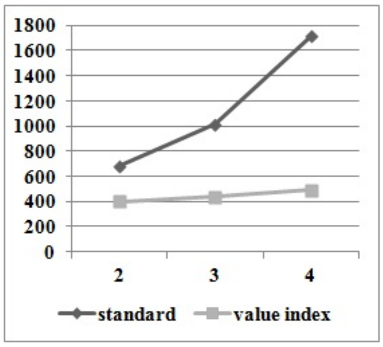
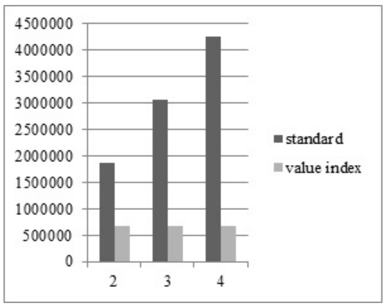


Figure 8: The query used in the experiment of changing the number of branches



(a) The elapsed time in milliseconds



(b) The total number of scanned elements

Figure 9: Varying the number of branches

6 Conclusion and Future Work

This paper proposed the Structural index to handle the structural part of CAS queries and the Content index to handle the content part. The indices utilized the semantics of XML schema and XML data in their construction. In addition, this paper introduced the query processing algorithms on the proposed indices. The performance evaluation proves the benefits gained from applying the semantic-based indices in trimming the search space and avoid unnecessary data scanning. The evaluation results on different XML datasets indicate that our proposed method provides a performance improvement by applying the semantic concepts in its Content index over the non semantic index. Due to space limitation, we are not able to describe the accuracy of our proposed indexing scheme in details within this paper. However, we can report that based on the data sets that we have used in this paper, the accuracy of the query results is approximately 97%. The details of this accuracy experiments by calculating the precision and recall will be in the extended version of this paper. Our future work will include other type of predicates. In fact, Boolean predicates are an important part of the query. Since they have not been widely investigated, in the next work, we would like to focus on this part of the queries especially when all the Boolean operators, i.e. AND, OR, NOT, come in a single query.

References

Liang, Z.P Han, J.Y. &Qian, G.(2006)'A multiple-depth structural index for branching query',*Information and Software Technology* **12**, 928–936.

Haw,S. &Lee, C. (2009)'Extending path summary and region encoding for efficient structural query processing in native XML databases'. *JSS, Elsevier Inc.*

Wu, H. Ling, T. W. Chen, B. &Xu, L.(2011)'TwigTable: Using Semantics in XML Twig Pattern Query Processing', *Journal on Data Semantics XV* pp.102–129

Chen, Z. Gehrke, J. Korn, F. Koudas, N. Shanmugasundaram &J. Srivastava, D. (2007)'Index structures for matching XML twigs using relational query processors', *Data & Knowledge Engineering* **60**, 283–302

Alghamdi, N. S. Rahayu, W. &Pardede, E. (2011), Object-Based methodology for XML Data Partitioning (OXDP), *in* 'The 25th IEEE AINA', Singapore, pp. 307–315.

Chung,J.K. Min, C.W. &Shim, K.(2002), APEX: an adaptive path index for XML data. *in* 'Proceedings of ACM SIGMOD', pp 121–132.

Goldman, R. &Widom, J.(1997), Dataguides :enabling query formulation and optimization in semistructured databases.*in* ' Proceedings of the 23rd VLDB', Athens, Greece, pp. 436–445.

Wang, H.S. Park, W. Fan, &P.S. Yu.(2003), ViST: a dynamic index method for querying XML data by tree structures.*in* ' ACM SIGMOD', pp. 110–121.

Tatikonda, S. Parthasarathy, S. &Goyder,M. (2007)LCS-Trim: Dynamic programming meets XML indexing and querying. *in* ' VLDB Endowment', ACM, pp. 63–74.

Quanzhong, Li. &Bongki, M.(2001)Indexing and querying XML data for regular path expressions. *in* 'Proceedings of 27th VLDB', Rome, Italy, pp. 361–370.

Rizzolo, F. &Mendelzon, A.O.(2001)Indexing XML Data with ToXin. *in* 'Proceedings of 4th WebDB'.

Zou, Q. Liu, S. &Chu, W. (2004)Ctree: A Compact Tree for Indexing XML Data. *in* 'WIDM04', Washington, DC, USA.

Monjurul Alom, B.M. Henskens,F. &Hannaford, M.(2009)Querying Semistructured Data with Compression in Distributed Environments.*in* 'the 6th Int. Conf. on Inf. Tech.: New Generations'.

University of Washington XML Repository, 2002. `http:// www.cs.washington.edu/research/xmldatasets`

XML Schema Part 0: Primer Second Edition, 2004. `http:// www.w3.org/TR/xmlschema-0/`