2013 IEEE 27th International Conference on Advanced Information Networking and Applications

Object-Based Semantic Partitioning for XML Twig Query Optimization

Norah Saleh Alghamdi, Wenny Rahayu and Eric Pardede
Department of Computer Science and Computer Eng
LaTrobe University
Melbourne VIC 3083 Australia
Email:{nalghamdi@students, w.rahayu@,e.pardede@}latrobe.edu.au

***Abstract*—The increased deployment of the XML-based standard for representation and exchange in multi-disciplinary domains has enforced the need for a more effective way to deal with XML query processing. Since very limited attention has been given to the semantic nature of the XML data being processed, this paper focuses on a technique for XML query optimization, called Object-based Twig Query (OTQ), to utilize the semantic structure of the data being queried to process twig queries. A twig query, which is a type of query with multiple branches, requires complex processing due to the joins between multiple paths. OTQ performs object-based data partitioning, which aims at leveraging the notion of frequently-accessed data subsets and putting these subsets together into adjacent partitions. It evaluates branched queries through two essential components: (i) OTQ indexing, which uses an object-based connection to construct its indices i.e. Schema index and Data index; and (ii) OTQ processing to produce the final results in optimal time. At the end of this paper, a set of experimental results for the proposed approach on a range of real and synthetic XML data, as well as a comparative study of a similar work in the area, is presented to demonstrate the effectiveness of OTQ optimization.**

***Index Terms*—XML; indexing; processing twig queries; high performance; query optimization; object-based partitioning; XML databases.**

I. Introduction

The powerful ability of XML in describing and presenting the data has been recognized as the standard of electronic data interchange in multi-disciplinary domains. The metadata in XML documents gives them a semantically rich structure which can be leveraged for various information system applications. It also opens up opportunities to improve techniques to access and process XML data. In this paper, we focus on processing XML queries efficiently by taking into consideration the semantics and structure of the underlying XML documents. In particular, we will focus on XML twig queries. A twig query is a type of query which accesses XML trees with multiple branches and requires complex processing due to the joins between multiple paths.

Different XML data indexing and query processing approaches have been proposed to support twig queries. The previous XML data indices were classified into three categories [1]. The first is Path-based indices such as APEX[2] and MDFB [3], which group nodes in data trees based on local similarity and have an adjustable index structure depending on the query workload. These indices need to deal with a huge index size because its index keeps tracks of the forward and backward paths to establish an effectively supportive layer to answer twig queries. The second is Node-based indices, such as TwigX-Guide [4], which index the position of each node within the XML tree and then process the nodes by joining them when in some cases; aggressive joins deteriorate the query performance. The third is Sequence-based indices such as ViST [5], PRIX [6] and LCS-Trim [7], which evaluate queries based on sequence matching after transforming both XML data and twig queries into sequences. However, the occurrence of false positives caused by sequence matching, which is not tree-matching, is the drawback of the third approach. In spite of these efforts, none of these works has considered the importance of utilizing the notion of objects of XML data in its index.

Utilizing XML schema during the construction of XML data index will assist in reducing frequent update of the index. Furthermore, in most cases, an XML schema presents all possible occurrences of the data structure of its instances, thus it is regarded as a highly descriptive resource for the current data and its updated versions.

Due to XML data and its schema being self-described, OTQ exploit this advantage to construc two types of indices namely Schema index and Data index. Incorporating the notion of objects of XML data into OTQ reduces a dilemma posed in previous XML indices which is increased size of indices to be close to the size of the data itself [8]. In this paper, we tackle the issue of iteration on a large index to find matched trees, or matched sequences by introducing Schema index. The Schema index usually has a smaller index size compared to an index built only based on the data. Data index in OTQ indexing approach also employs the the notion of objects of the indexed data by grouping the data within objects. Introducing the idea of objects accelerates processing queries by localizing each of them into a small portion of data.

OTQ incorporates the semantic features of the index construction into the twig query processing approach for an efficient pruning technique which starts by trimming the search space at the Schema index before drilling down to the Data index. The object-based intersection accomplishes this goal to find the participating paths and objects. The pruning technique avoids unnecessary searching through irrelevant portions of the Data index by eliminating non-participating paths and objects.

1550-445X/13 $26.00 © 2013 IEEE
DOI 10.1109/AINA.2013.74

The key contributions of this paper are summarized as follows:

1) Exploiting the semantics of XML schema and data in the construction design of the OTQ index.
2) Introducing an object-based intersection technique, which trims the search space based on the knowledge of structures and contents derived from XML schema.
3) Eliminating irrelevant portions of data at the Data index by discarding unmatched paths and irrelevant objects.

The remainder of this paper is organized as follows. A survey of related work is presented in section 2. We present an overview of the system architecture in section 3. Thereafter, in section 4, we describe in detail the OTQ algorithms. The experiments and evaluation are presented in section 5. We conclude our work in section 6.

II. RELATED WORKS

Several different approaches have been taken in the previous literature to index XML data and processing twig queries [9], [4], [5], [6], [10], [3]. Haw and Lee [1] divided earlier works on indexing XML data into three main groups: Path-based indices, Node-based indices and Sequence-based indices.

The Path-based indexing approach relies on creating a path summary for XML data beginning from the root to a particular node such as DataGuide [9]. The size of this index tends to be huge and does not support branch queries. MDFB [3] is a cover index which covers all possible forward and backward paths to answer twig queries. To reduce the size of the index, APEX[2] groups XML data nodes by considering their local similarity, having the ability to adjust its structure based on query workloads. In our recent work, OXiP [11] tokenizes all rooted label paths in an XML schema and preserves the pathways within each object partition of XML data to answer path queries in optimal time.

In Node-based indexing, the structural information of XML data is captured using a certain labeling scheme. [12] survey the different proposed numbering/labeling schemes in earlier work. Tree pattern matching is then performed on a labeling scheme. [13] benchmarks the different performance achieved by the proposed approaches to process XML twig patterns by navigating and joining the labeling scheme. Haw and Lee [4] proposed TwigX-Guide which decomposes a twig query into a set of parent-child and ancestor-descendant relationships. The match-merge process is used to combine these decomposed relationships. However, the essential shortcoming of TwigX-Guide is the increase in the unnecessary intermediate results, especially when the size of the final results escalates or the degree of branches in the twig query increases.

Transforming XML data and a twig query into sequences and then performing subsequence matching is the approach of Sequence-based indices to solve the drawbacks of previous approaches. ViST [5] uses the Virtual Suffix Tree to encode the data and the query but the main drawback of this approach is the incidence of false alarms because of sequence matching. PRIX encodes the sequence using the prüfer labeling system which constructs a one-to-one correspondence between trees and sequences. PRIX [6] solves the problem of false-positives by introducing a complex four-phase to refine the result. However, the proposed solution is time consuming since it is done by document post-processing [14].

Bruno et al. in [10] used a holistic twig join algorithm TwigStack to avoid producing large intermediate results. Jiang et al. addressed the problem of efficient processing of holistic twig joins on all/partly indexed XML documents using the holistic twig join algorithms [15]. Chen et al. in [16] proposed a Twig2Stack algorithm which uses hierarchical-stacks but due to its complexity of hierarchical-stacks, it may conduct many random accesses and may use a large memory space.

Despite the earlier works, reducing the index size remains an open issue. To overcome this limitation, we propose a new methodology that keeps the index to a reasonable size, built based on the schema first, then utilizing the knowledge of objects to accelerate the query process on only a filtered portion of Data index. None of the existing approaches incorporate the advantage of XML being self-described into constructing their index which can certainly improve the structural query time performance.

III. THEORETICAL FRAMEWORK OF OTQ

In this section, we describe a new index method based on the concept of objects in XML structures to increase twig query performance, called Object-based methodology for Twig Query index (OTQ). In the stage of parsing the schema, the concept of objects of our earlier work, OXDP [17], is utilized and leveraged to develop the OTQ index as a pre-processing stage. Basically, OXDP extracts objects from a given XML schema and its validated data. Following our earlier work, in this paper, we focus on optimizing index construction and processing twig query. As can be seen in Fig. 1, the system goes through two important stages: (i) OTQ indexing where the system leverages the notion of objects to construct its indices, namely Schema index and Data index; and (ii) OTQ processing to process a user-query. The OTQ system shows that the notion of objects can be utilized to create an XML index of a reasonable size and shows how this concept can significantly trim the search space to gain better performance for twig queries. Fig. 1 is an overview of the OTQ system.

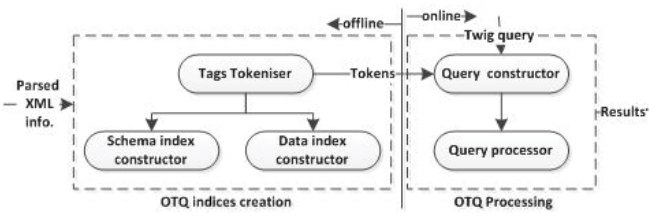

Fig. 1: OTQ System Overview.

A. OTQ indices

OTQ indices, Schema index and Data index, are used to evaluate Twig queries. XML schema is used since it contains descriptions of XML data nodes and their constraints and

relationships. In addition, building an index based on XML schema will reduce the need to update the index when the XML data instances change without any change in the schema. Both these types of indices are constructed based on three aspects: (i) Twig-aspect, which preserves the details of twig patterns and is represented by Schema Object and Data Object in our proposed method; (ii) Path-aspect, which preserves the details of a twig pattern of a path and is represented by the path of Schema Object and the path of Data Object; and (iii) Object-aspect, which partitions and links the data based on the XML objects which are represented as definition 1.

Definition 1: (Object): An object is defined as an element of XML schema with complexType or complexContent and it is a non-leaf node in the schema. Objects might consist of other nested objects and basic elements.

B. Schema Index

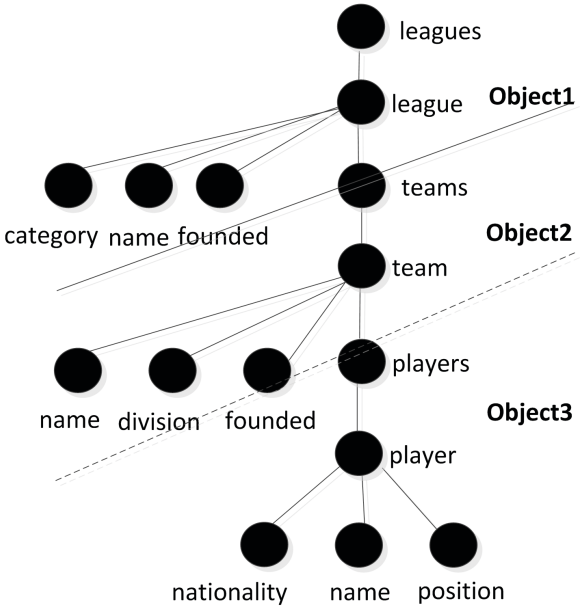

Fig. 2: Leagues Schema Tree.

TABLE I: Schema index for the Schema of Leagues

Tokenized Tags			
Tags	Tokens	Tags	Tokens
leagues	T0	team	T6
league	T1	division	T7
category	T2	players	T8
name	T3	player	T9
founded	T4	nationality	T10
teams	T5	position	T11

Schema Object	
S_o	Tokens
S_o1	T0 T1
S_o2	T1 T2 T3 T4 T5
S_o3	T5 T6
S_o4	T6 T3 T7 T4 T8
S_o5	T8 T9
S_o6	T9 T3 T10 T11

Path of Schema Object		
Path	Schema Object	Objects
P1	S_o1 S_o2	Obj1
P2	S_o1 S_o2 S_o3 S_o4	Obj2
P3	S_o1 S_o2 S_o3 S_o4 S_o5 S_o6	Obj3

In this section, the basic definitions used throughout this paper related to Schema indices are presented. Fig. 2 shows the schema tree for Leagues which has information about sport leagues and consists of three object partitions as outputs of OXDP[17]. The object partitions in Fig.2 are Objcet1 (leagues (league (category, name, founded))), Object2 (teams (team (name, division, founded))), and Object3 (players (player (nationality, name, position))).

Definition 2: Tokens(T): A token is a unique value that encrypts each distinct tag name of XML schema.

Example 1: In Table 1 , the Tokenized Tags table consists of all distinct tags of the Leagues XML Schema along with their tokens. In the same table, "name" has T3 as its token, T0 represents the token of "leagues" and T9 is the token of "player".

Definition 3: Schema Object (S_o): Schema Object is a complex element of an XML Schema, represented by a tree with two levels combining a tag of complex elements as a root and tags of its children's leaves. Consider S_o as the set of Tokens T(T_{root}, T_1,...,T_k), where T_{root} is the token of the complex element tag or name, T_1 is the token of the first child node of the root and T_k is the token of the last child node where k is the number of the root's children.

Example 2: In Table 1, S_o2 in the Schema Object table maintains the tokens of the tags at the Twig-aspect by connecting the root "league" with its children ("category", "name", "founded", "teams") represented by tokens T1, T2, T3, T4 and T5. Only distinct S_o is stored in the Schema Object table of Table 1.

Definition 4: Path of Schema Object (P): Path of Schema Object is a set of S_o located on the same path from the XML schema's root to a leaf node.

Example 3: Consider Path of Schema Object in Table 1, P1, which consists of S_o1 and S_o2, can represent the left most path of the leagues' schema tree from the root element "leagues" to the leaf element "category", including the element "league".

C. Data Index

In this section, the basic definitions used throughout this paper related to Data index are presented. Table 2 and Fig. 3 show the indices' structure beside Leagues XML data which is a valid data with Leagues XML schema in Fig. 2.

Definition 5: 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 (T_{parent}, $T_{child(1)}$, ..., $T_{child(k)}$, Pos_{parent}, $Pos_{child(1)}$, ..., $Pos_{child(k))}$ where T_{parent} is the element token of the parent node, $T_{child(i)}$ 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.

Example 4: Consider D_o6 in Data Object-object3 in Table 2 as an example of Data Object for S_o6 (T9 T3 T10 T11), which corresponds to the twig pattern ("player", "name", "nationality", and "position"). The value 12, 13, 14 and 15

are identifiers of the XML data position of D_o6. Each D_o of Data Object representing a specific S_o of Schema Object should contain the same sequence of tokens determined by the S_o.

Definition 6: Path of D_o (dp): Path of D_o is a sequence of D_o which takes a place on the same path from the root element of the current object until the last D_o presents on this XML path.

Example 5: From Table 2, dp1 has D_o1, D_o2 as a sequence of Data Objects allocating in the same path which is P1.

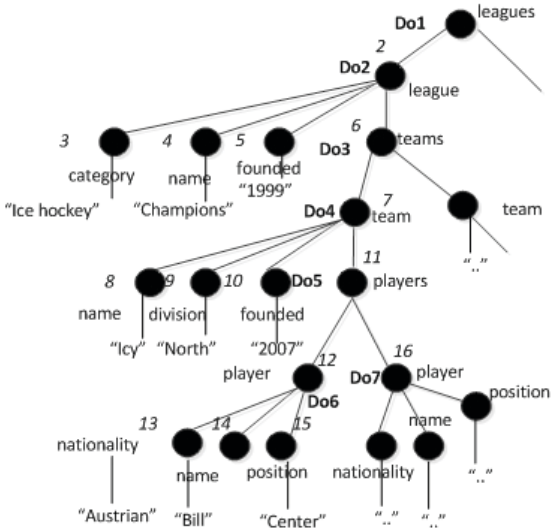

Fig. 3: Leagues XML data

TABLE II: Data index for the data of Leagues

Data Object - object 1-		
D_o	Tokens	XML data position
D_o1	T0 T1 ..	1 2 ..
D_o2	T1 T2 T3 T4 T5	2 3 4 5 6
Data Object - object 2-		
D_o	Tokens	XML data position
D_o3	T5 T6 ..	6 7 ..
D_o4	T6 T3 T7 T4 T8	7 8 9 10 11
Data Object - object3-		
D_o	Tokens	XML data position
D_o5	T8 T9 ..	11 12 ..
D_o6	T9 T3 T10 T11	12 13 14 15
Path of Data Object		
dPaths	Path of S_o	Path Of D_o
dP1	P1	D_o1 D_o2
dP2	P2	D_o3 D_o4
dP3	P3	D_o5 D_o6

IV. OTQ PROCESSING

This section details the algorithms that process queries based on the concept of objects. It shows how a twig query is constructed to build the structure of the root and children. Thereafter, it presents OTQ algorithms and how they are interconnecting with each other.

A. Query Construction

OTQ index construction takes place before the actual commencement of query processing. The TreeQueryBuilder method is invoked when a twig query is started. TreeQueryBuilder is a simple method used to analyze a user-entry query into nodes and identify parent and children nodes. A twig query is constructed into query nodes each of which consists of a root tag, a root token and its children's tags and tokens. Twig query nodes are presented in the OTQ algorithm following definition 7.

Definition 7: Twig query nodes (Tqn) are a set of nodes associated with their token and can be represented as Tqn(n_{root}, n_1, ..., n_k, T_{root},T_1,...,T_m), where n_{root} is the twig root tag, each n_i is a child's tag of n_{root}, k is the number of children in this twig and each T_j is the token of the node n_i and m is the number of children's token.

B. OTQ Algorithms

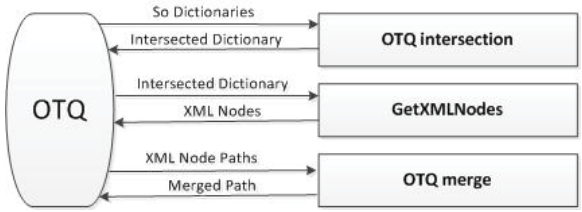

Fig. 4: OTQ Algorithms.

The query nodes resulting from the query builder are processed through OTQ, OTQ intersection, GetXMLNode and OTQ merge functions to produce the final result (see Fig. 5). The OTQ method is the main method in our algorithm because it is connected with other functions to perform the query. The process starts with validation of the Paths of Schema Object of the query root based on the object on which the paths end. In order to speed up OTQ process further, we want to avoid a large number of joins in the Schema index to retrieve the targeted path of Schema Object; Schema Object Dictionary (S_oDic) in definition 8 is created to link Tokenized Tags, Schema Object and Path of Schema Object as an optimized step in processing XML Schema indices. S_oDic reduces the number of Tokens, S_o and Path of S_o required for joining process. It is achieved by pruning S_oDic based on the intersection of object IDs.

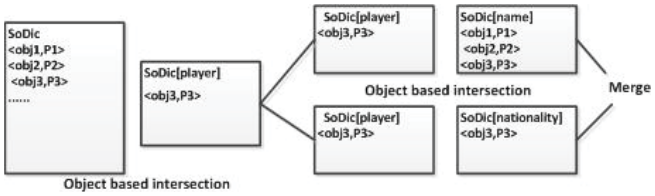

Fig. 5: Object-based Intersections for Q = "player[./name] [./nationality]".

```
Input: SoDic , SoDictionary
Output: iRs
1:  for all SoD in SoDic do
2:    if ! SoDictionary.Contains(SoD.object) then
3:      continue;
4:    end if
5:    iRs.Add(SoD.object );
6:    temp=SoDictionary[object];
7:    for all So in SoD.PathOfSo do
8:      if temp.Contains(So) then
9:        iRs[object].Add(So);
10:     end if
11:   end for
12: end for
13: return iRs
```

Fig. 6: OTQ intersection.

Definition 8: Schema Object Dictionary (S_oDic) is an optimized dictionary for the Schema indices used in processing the query and consists of a combination of Tokenized Tags, Schema Object and Path of S_o. It link each token with all possible Paths of S_o and the object on which each path ends.

A Twig query is decomposed into paths and S_oDic of the root with S_oDic of the whole data are intersected using OTQ intersection method as in Fig. 6 . This scenario handles the cases of parent-child (P-C) and ancestor-descendant (A-D) edges inside the twig query as well as the links between objects. In Fig. 5, the process of a query: player[/nationality][/name] starts by searching inside S_oDic using the root of the query, which is "player". The query is split into "player/nationality" and "player/name" and each of S_oDic of those children is intersected with S_oDic of the query root. This will speed the query process up by trimming the search space (see OTQ intersection in Fig. 6). In GetXMLNode in Fig. 8, a list of path of D_o that share the same Path of S_o is used to find the query results. the algorithm goes through each D_o in the participated object to find a matched token of data elements to a query token. However, this function assists in eliminating non-participating paths and objects, thus, the search space is reduced. Then OTQ merge function will be invoked to merge the decomposed paths (see Fig. 7).

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 S_o and Path of D_o, and (ii) partition and keep links between interconnected data based on object.

In addition, object-based intersection assists in preserving the correctness when linking the object partitions. For instance, Fig. 9 shows a portion of league schema from Fig. 2, a track of all siblings with the same parent is kept ("teams", "team"), "(team", "name", "founded")("players", "player", "player"), and a track of all twigs in the same path from the root to the leaf ("teams", "team")("teams", "team",

```
Input: P1, P2, depth
Output: P
1:  for all (e1 in P1) do
2:    for (i=0 to depth) do
3:      prefix.Add(e1[i])
4:    end for
5:    for (i=depth+1 to e1.Count) do
6:      sufix.Add(e1[i])
7:    end for
8:    if (!tmp[prefix]) then
9:      tmp[prefix].Add(sufix)
10:   end if
11: end for
12: for all (e2 in P2) do
13:   for (i=0 to depth) do
14:     prefix.Add(e2[i]);
15:   end for
16:   for (i=depth+1 to e2.count) do
17:     sufix.Add(e2[i]);
18:   end for
19:   if (tmp[prefix]) then
20:     for all (e1 in tmp[prefix]) do
21:       Tr = prefix;
22:       Tr.Add(e1);
23:       Tr.Add(sufix);
24:       if (Tr.Count = Count(prefix+e1+sufix)) then
25:         P.Add(prefix);
26:         P[P.Count].Add(e1);
27:         P[P.Count].Add(sufix);
28:       end if
29:     end for
30:   end if
31: end for
32: return P;
```

Fig. 7: OTQ merge.

"players") where players is a nested object of team. If the query "team[/name][//player]" is executed, the same path of S_o which is P3 will connect the two objects in players, as explained above.

OTQ leverages the advantage of the semantic workload of queries during the construction of its indices. This characteristic using XML data partitions coupled with an efficient linking technique among partitions to process queries will have a significant impact on the performance of XML queries. From this point of view, it is known that not all parts, called objects in this paper, of XML data are equal in "access rate"; some objects are more frequently used than others. Therefore, it is obvious that the "access rate" to some index nodes is highly likely to vary, because of their relativity to the position of the index structure. OTQ utilizes the object-based intersection in addition to its ability to discard irrelevant objects and the path of D_o to reduce query response time.

```
Input: P, Qp;
Output: Xns;
1: Paths = PathsOfD_o[p];
2: for all (dp in Paths) do
3:   D_os = Path of D_o in the current dp;
4:   k = 0;// to track the query elements.
5:   for (j=0 to Dos.Count) do
6:     D_o= D_os[j];
7:     for (object in Data Object) do
8:       // only participated obj
9:       for all (r in object[D_o]) do
10:        for (i=0 to r[Token].count & k<Qp.count) do
11:          if (r[Token][i]= Qp[k].Token) then
12:            TmpRes.Add(obj[XPos][i]);
13:            if (k > 0 & Qp[k].Relationship ! = A-D)
               then
14:              TmpRes.Remove(TmpRes.count);
15:              continue;
16:            end if
17:          end if
18:          if (Qp[k].Children.count = 0) then
19:            for (++i; i < r[Token].count;i++) do
20:              if (r[Token][i] = Qp[k].Token) then
21:                tmp = TmpRes;
22:                tmp[tmp.Count] = r[XPos][i];
23:                Xns.Add(tmp);
24:              end if
25:            end for
26:          end if
27:          k++;
28:          if (k=Qp.count & j+1<D_os.Count) then
29:            if (TmpRes.count=Qp.count) then
30:              Xns.Add(TmpRes);
31:              TmpRes.Remove (TmpRes.count);
32:            end if
33:            k–;
34:          end if
35:          if (k=Qp.Count or Qp[k].Children! =0) then
36:            break;
37:          end if
38:        end for
39:      end for
40:    end for
41:  end for
42:  if (TmpRes.count=Qp.count) then
43:    Xns.Add(TmpRes);
44:  end if
45: end for
46: return Xns;
```

Fig. 8: GetXMLNodes.

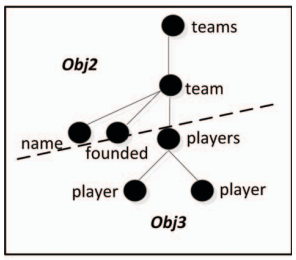

Fig. 9: Linking Technique.

V. EXPERIMENTAL RESULTS

A. Experimental Environment Set Up

In this section, the performance of our algorithms is compared with the state-of-the-art TwigX-Guide [4]. TwigX-Guide has outperformed other Twig querying systems, taking advantage of both the path summary in DataGuide[9] for efficient path queries with parent-child edges and the region encoding in TwigStack [10] in its ability to process twig queries. The performance of TwigX-Guide outperformed other well known methods, which are TwigStack [10] , TwigStackXB[10] , TwigINLAB [18] and TwigStackList [19], in most queries for the comparsion done by Haw et al [4].TwigX-Guide is not publicly available at this time; it has been implemented based on the algorithm described in [4]. All the algorithms of our proposed method were implemented using C Sharp and the experiments were run on a 3.2GHz Intel(R) Core(TM) i5 machine with 6 GB memory running Windows7.

B. Data Sets

The datasets used in the experiment are shown in Table 3. These datasets were obtained from the University of Washington's XML repository [20]. The chosen datasets differ in their characteristics. DBLP dataset has a shallow structure with a recursion in some of its element names. The document trees in the SigmodRecords and Yahoo datasets were used due to their nested structure. Additional information is provided in Table 4, including the maximum depth, the number of elements, and the number of object partitions for each dataset.

C. Evaluation Criteria

The performance of OTQ is compared with TwigX-Guide based on the type of relationships (P-C, A-D, or mixed). The average query processing times for a given twig query is calculated. As mentioned earlier, the query processing time is the time taken by the OTQ processing algorithm excluding the off-line stage i.e. building Schema index and Data index.

TABLE III: Datasets for the experiment.

Dataset	Size(Kbytes)	No.Elements	Max-depth	No.Objects
SigmodRecord	466	19908	6	2
Yahoo	1,303	30210	5	2
DBLP	130,582	6337977	6	8

TABLE IV: Queries for the experiment.

Q	Queries	#Obj	Depth
QS1	SigmodRecord/issue[/volume][/number]	1	3
QA2	root/listing/auction_info/high_bidder [/bidder_name][/bidder_rating]	1	5
QA3	root/listing/auction_info[/current_bid] /time_left	1	4
QD4	/dblp/mastersthesis[/title]/author	1	3
QD5	/dblp/book[/ee]/year	1	3
QD6	//phdthesis[/year][/series][/number]	1	4
QS7	articles[//title][//author]	2	7
QS8	issue[//volume][//author]	2	7
QD9	//inproceedings//title[//i][//sub][//sup]	1	5
QD10	//inproceedings[//month][//url][//ee]	1	4
QD11	//inproceedings[//month][//url][//ee]//title	1	4
QS12	SigmodRecord/issue[/volume][//title]	1	5
QS13	article[/title][//authors]	2	7
QD14	//inproceedings[//title/i][//year]	1	4
QD15	/dblp/article[/journal][//sup]	1	3
QD16	/dblp/incollection[/booktitle][//ee]	1	3
QD17	//article/title[//i][//sub]	1	4

Similar to OTQ, TwigX-Guide considers its index construction an off-line stage [4].

D. Queries

Table 4 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. These queries have different characteristics in terms of the number of access objects, depth of the path, type of edge relationship and twig structure.

All the queries of the first group have only P-C edges connecting their nodes and need to access only a single object. QS1 to QD6 are twig queries with different number of nodes. Their branches vary from 1 to 3. The query answer will be retrieved from a single object.

In the second group of queries, QS7-QD11 have only an A-D relationship between their nodes. Some need to access one object partition and others needs to access two object partitions to retrieve the data. The maximum depth of retrieved data is 7 in QS7 and QS8, 5 for QD9 and 4 for QD10 and QD11.

The third group of queries, QS12 - QD17, concerns the performance of the twig queries with Mixed edges which is the Hybrid type of relationship connecting between nodes (P-C and A-D). It also consists of a different number of nodes and branches. QS13 needs to access two objects to get the final results. However, the rest need to access one object to retrieve the answer of the query.

VI. Performance Analysis

We analyze the performance by varying the type of relationship between query nodes.

A. Only P-C Twig Queries

Despite the strength of TwigX-Guide in processing and optimizing P-C twig queries, OTQ outperforms TwigX-Guide in processing these types of queries, as shown in Fig. 10. For instance in QS1, TwigX-Guide decomposes the query into two paths SigmodRecord/ issue[/volume] and SigmodRecord/issue[/number]. The answer to each path is retrieved directly from DataGuide and it only needs one join to yield the final result. However, OTQ is able to trim the search space in the schema index as well as in the data index. OTQ gains advantage from its index construction since it preserve the details of all objects located in each path of the schema and data as in Path of S_o and Path of D_o. This feature supports our approach to handle P-C queries more efficiently. Fig. 10 shows how OTQ outperforms TwigX-Guide when querying P-C twig queries in SigmodRecords, Yahoo and DBLP datasets. The average improvement over all the queries with P-C edges is 44.36%.

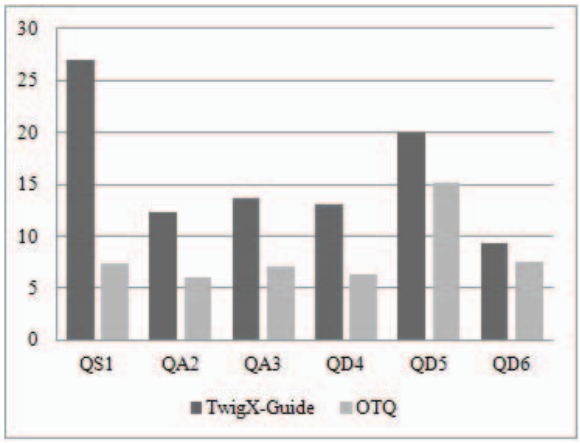

Fig. 10: The performance of P-C twig queries in milliseconds.

B. Only A-D Twig Queries

In this case, TwigX-Guide retrieves qualified data from A-D queries based on region encoding labeling. It incurs a large number of joins that causes a significant delay. OTQ has the ability to skip large portions of Schema Object through object-based intersection as well as discarding irrelevant objects and paths from Data Object. Thus, OTQ performs significantly better than TwigX-Guide for A-D queries, as shown in Fig. 11. The average improvement over all the queries with A-D edges is 78.05%.

C. Mixed Edges Twig Queries

TwigX-Guide performs better in P-C compared to A-D and performs better in mixed edges compared to A-D, since it uses path matching from Dataguide instead of node matching which reduces the number of joins that need to happen only on A-D. OTQ outperforms TwigX-guide in this group of queries as depicted in Fig. 12. OTQ performs significantly better than TwigX-Guide for queries with hybrid edges by around 88.87%. Overall average improvement of the evaluation times for all experimental queries is around 68.76%.

VII. Conclusion and Future Research

In this paper, a new technique was proposed, OTQ, to evaluate and process twig queries in XML data. Twig queries

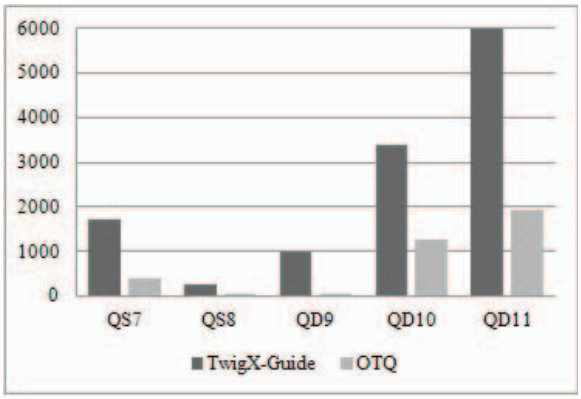

Fig. 11: The performance of A-D twig queries in milliseconds.

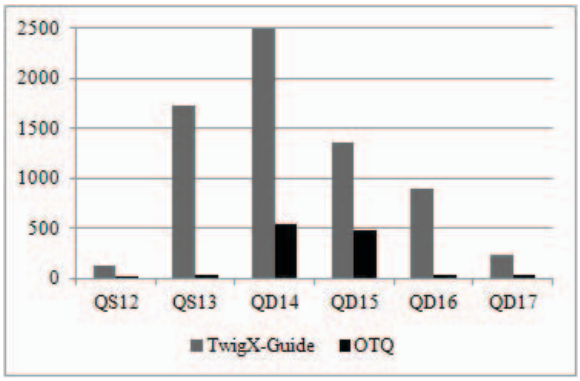

Fig. 12: The performance of mixed edges twig queries in milliseconds.

have been considered the most complex part of query processing over XML data due to the complexity of traversing data and finding matched results. OTQ utilizes the advantage of XML data being self-described to enhance the notion of a frequently-accessed data subset by employing a Object-based data partition. OTQ maintains its indices i.e. the Schema index and Data index in three aspects: the Twig-aspect, Path-aspect and Object-aspect to rapidly process queries. The Schema index is manipulated using the object-based intersection process to trim the search space and avoid unnecessary joins. Irrelevant data scanning is discarded in the Data index by eliminating irrelevant objects and paths. Our experiment shows that OTQ outperforms TwigX-Guide when we compare their performance on different path edge types i.e. (P-C, A-D, or mixed). The calculated average query processing times for a given query indicates that OTQ is a viable solution for processing Twig queries. We have mentioned that OTQ utilizes the schema which incorporates all possible structures of data in an application, even for non-materialized data. Therefore, the number of index updates will be reduced since the Schema index will outfit most future updates. In future research, we will need to investigate how to maintain the indices. Moreover, since OTQ considers processing queries without predicates, we will explore how to process complex queries with different types of predicates.

VIII. Acknowledgment

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

References

[1] S. Haw and C. Lee, "Data storage practices and query processing in XML databases: A survey," *Knowledge-Based Systems, Elsevier B.V.*, 2011.

[2] J. Chung, C.W. Min and K. Shim, "APEX: an adaptive path index for XML data," in *Proceedings of ACM SIGMOD*, 2002, pp. 121–132.

[3] L. Z. Han, J.Y. and G. Qian, "A multiple-depth structural index for branching query," *Information and Software Technology*, vol. 48, pp. 928–936, 2006.

[4] S. Haw and C. Lee, "Extending path summary and region encoding for efficient structural query processing in native XML databases," *JSS, Elsevier Inc.*, 2009.

[5] H.Wang, S. Park, W. Fan, and P. Yu, "ViST: a dynamic index method for querying XML data by tree structures," in *ACM SIGMOD international conference on Management of data*, 2003, pp. 110– 121.

[6] P. Rao and B. Moon, "PRIX: indexing and querying XML using prufer sequences," in *Proceedings of ICDE*. IEEE, 2004, pp. 288–300.

[7] S. Tatikonda, S. Parthasarathy and M. Goyder, "LCS–Trim: Dynamic programming meets XML indexing and querying," in *VLDB Endowment, ACM*, 2007, pp. 63–74.

[8] S. Abiteboul, P. Buneman, and D. Suciu, *Data on the Web: From Relations to Semistructured Data and XML*. Los Altos, CA 94022, USA: Morgan Kaufmann Publishers, 1999.

[9] R. Goldman and J. Widom, "Dataguides :enabling query formulation and optimization in semistructured databases," in *Proceedings of the 23 International Conference on VLDB, Athens, Greece*, 1997, pp. 436–445.

[10] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: optimal xml pattern matching," in *Proceedings of ACM SIGMOD*, 2002, pp. 310–321.

[11] N. Alghamdi, W. Rahayu, and E. Pardede, "OXDP and OXiP: The notion of objects for efficient large XML data queries," *IJGUC, Inderscience*, Accepted 12 September 2011.

[12] S. Haw and C. Lee, "Node labeling schemes in XML query optimization: a survey and trends," IETE Technical Review 26, 2009.

[13] J. Lu, "Benchmarking holistic approaches to XML tree pattern query processing," in *DASFAA*. LNCS 6193, Springer-Verlag Berlin Heidelberg, Japan, 2010, pp. 170–178.

[14] S. Mohammad and P. Martin, "XML structural indexes," Queen University, Technical report 560, 2009.

[15] H. Jiang, W. Wang, H. Lu, and J. X. Yu, "Holistic twig joins on indexed xml documents," in *In Proc. of VLDB*, 2003, pp. 273–284.

[16] S. Chen, H. gang Li, J. Tatemura, W. pin Hsiung, D. Agrawal, and K. S. C, "Twig2stack: bottom-up processing of generalized-tree-pattern queries over xml documents," in *In Proc. of VLDB*, 2006, pp. 283–294.

[17] N. Alghamdi, W.Rahayu, and E. Pardede, "Object-Based methodology for XML Data Partitioning (OXDP)," in *The 25th International Conference on Advanced Information Networking and Applications , Singapore*. IEEE, 2011, pp. 307 – 315.

[18] S.-C. Haw and C.-S. Lee, "Stack-based pattern matching algorithm for xml query processing," *Journal of Digital Information Management*, 2007.

[19] J. Lu, T. Chen, and T. W. Ling, "Efficient processing of xml twig patterns with parent child edges: a look-ahead approach." in *Proceedings of CIKM*, 2004, pp. 533–542.

[20] (2002) University of Washington XML Repository. [Online]. Available: http://www.cs.washington.edu/research/xmldatasets