IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 30, 2020, accepted March 15, 2020, date of publication March 31, 2020, date of current version April 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2984600

Systematic Partitioning and Labeling XML Subtrees for Efficient Processing of XML Queries in IoT Environments

FARAG AZZEDIN, SALAHADIN MOHAMMED, MUSTAFA GHALEB, JAWEED YAZDANI, AND ADEL AHMED

Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

Corresponding author: Farag Azzedin (fazzedin@kfupm.edu.sa)

This work was supported by the King Abdulaziz City for Science and Technology (KACST) through the Science and Technology Unit at King Fahd University of Petroleum and Minerals (KFUPM), as part of the National Science, Technology and Innovation Plan, under Grant 11-INF1657-04.

ABSTRACT With the advent of IoT, storing, indexing and querying XML data efficiently is critical. To minimize the cost of querying XML data, researchers have proposed many indexing techniques. Nearly all the techniques, partition the XML data into a number of data-streams. To evaluate a query, existing twig pattern matching algorithms process a subset of the data-streams simultaneously. Processing many data-streams simultaneously results in some or all of the following four problems, namely, the accessing of many data nodes which don't appear in the final solution of a given query, the generation of duplicate results, the generation of huge number of intermediate results, and the cost of merging the generated intermediate results. To the best of our knowledge, all the existing twig pattern matching algorithms suffer from some or all of the above mentioned problems. This paper proposes a new twig pattern matching algorithm called MatchQTP which processes one data-stream at a time and avoids all the above mentioned four problems. It also proposes a new indexing technique called RLP-Index and a new XML node labeling scheme called RLP-Scheme, both of which are used by MatchQTP. Unlike the existing indexing techniques, RLP-Index stores a subset of the data nodes. The rest of the data nodes can be generated efficiently. This minimizes storage space utilization and query processing time and makes RLP-Index the first of its kind. Many experiments were conducted to study the performance of MatchQTP. The results show that MatchQTP is very efficient and highly scalable. It was also compared with four algorithms, three of which are used frequently in the literature to compare the performance of new algorithms and the fourth algorithm is the state-of-the-art algorithm. MatchQTP significantly and consistently outperformed all of them.

INDEX TERMS IoT, XML indexing, twig queries, XML query processing, tree-pattern matching, node labeling.

I. INTRODUCTION

With the advent of Internet of Things (IoT), more and more heterogeneous devices generate and transmit data in certain formats defined by manufactures or alliances [1], [2]. The vision of IoT results in connecting various smart devices. As stated in [2]–[4], the number of interconnected devices is expected to reach 20 billion by 2020. Therefore, it is important to efficiently process the huge data generated from heterogeneous devices. These devices are the data sources utilized by smart applications. The value of these IoT smart applications comes from the quality and the efficiency of the data generated by these devices. Data efficiency is critical and it presents a challenge for IoT applications due to the massive number of heterogeneous devices with specific characteristics. Furthermore, since the data is in different formats, it is critical to have a common data specification for message exchange. For the time being, there are two major languages for data exchange, namely eXtensible Markup Language (XML) and JavaScript Object Notation (JSON) [1], [2]. XML has more complete definition on data format. Furthermore, the Web Ontology Language (OWL) is

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Marozzo.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

a Semantic Web language developed to represent complex and yet rich knowledge about things and their relations. As such, the OWL format is the most used technique to describe sensor-generated data due to its ability to simplify both reasoning and semantic interoperability among smart devices [1], [2].

Unfortunately, the huge volume of XML data generated by IoT devices, makes the efficient storage and processing of XML data a necessity. As a result, a number of multi-model and native XML database management systems have emerged [5]. Furthermore, exchanging large datasets using a verbose language such as XML is not efficient [6]. In addition, IoT-generated data faces the issues of interoperability and reusability [7].

As we know that an XML dataset, denoted as $\mathbb{X}$, can be modeled as a tree, XML-tree, Figure 1a. A node in $\mathbb{X}$ (data-node) represents an element or an attribute or a value. An edge in $\mathbb{X}$ represents a parent-child relationship between two nodes.

XML queries are typically written in XPath [8] or XQuery [9] and they can also be represented as a small tree (a twig) pattern. A node in a twig pattern (Qnode) represents an element name or an attribute name or a value. An edge in a twig pattern represents a parent-child (PC) or an ancestor-descendant (AD) relationship between two Qnodes. Figure 1b is an example of a twig pattern. The single lined edges are PC edges whereas the double lined edges are AD edges.

The central issue in XML query processing is finding all the matches of a given query twig pattern (QTP) in $\mathbb{X}$; thus, many researchers have proposed a number of twig pattern matching algorithms (TPMAs) [10]–[12]. The cost of processing a QTP is the sum of the I/O and the CPU costs incurred to answer the QTP. Since the I/O cost of a QTP is much higher than its CPU cost, the overall cost of a QTP is approximated by its I/O cost.

To minimize the I/O cost of a QTP, many researchers have proposed a number of indexing techniques [10]–[12]. The techniques partition the data-nodes into a number of data-streams. To identify the structural relationship between data-nodes of different streams, many node-labeling schemes have been proposed [13], [14].

Existing TPMAs access a subset of the node-streams to answer a QTP. Generally, TPMAs who access fewer data-nodes incur less I/O cost and thus they are more efficient. Accessing multiple node-streams simultaneously, during QTP processing, generates duplicates and intermediate results. Duplicate removal and the merging of intermediate results are the two major components of the CPU cost of a QTP, and thus they must be minimized or avoided.

In this work we are proposing a new TPMA, a new indexing technique, and a new node-labeling scheme. Compared to the existing indexing techniques, the proposed technique partitions data-nodes into the highest number of data-streams. During QTP evaluation, the proposed TPMA reads only the data-streams that contain the output nodes, which reduces the I/O cost of the QTP significantly. Nearly all the existing TPMAs process many data-streams at a time; and as a result, they match data-nodes in different data-streams, generate duplicates and intermediate results. The proposed TPMA access only one data-stream at a time, and thus, it doesn't match data-nodes in different data-streams, and it doesn't generate duplicates or intermediate results. The main contributions of the proposed work are:

- A new node labeling scheme, called RLP-Scheme. The scheme is similar to the Dewey [15], but each node label contains more information than that of Dewey.
- A new indexing technique called RLP-Index, which partitions the data-nodes into the highest number of data-streams when compared to the existing approaches. In this technique, only a subset of the data-nodes are labeled and stored, the others can be efficiently computed. This is the first storage technique which uses this approach.
- A new TPMA that never accesses a data-node that is not part of the final solution, unless, the QTP has value predicates. It is the first TPMA that processes only one data-stream at a time to answer a QTP. To the best of our knowledge, it is also the first TPMA to avoid duplicate removal and the matching of data-nodes across multiple data-streams.
- Experimental results and performance analysis of the proposed TPMA.

The rest of the paper is organized as follows. Literature review is discussed in Section 2. The importance of XML data in IoT is detailed in Section 3. The proposed node labeling scheme and the proposed storage organization are outlined in sections 4 and 5 respectively. Section 6 presents the proposed TPMA while the performance analysis of the proposed TPMA is discussed in Section 7. Section 8 concludes the paper.

II. RELATED WORK

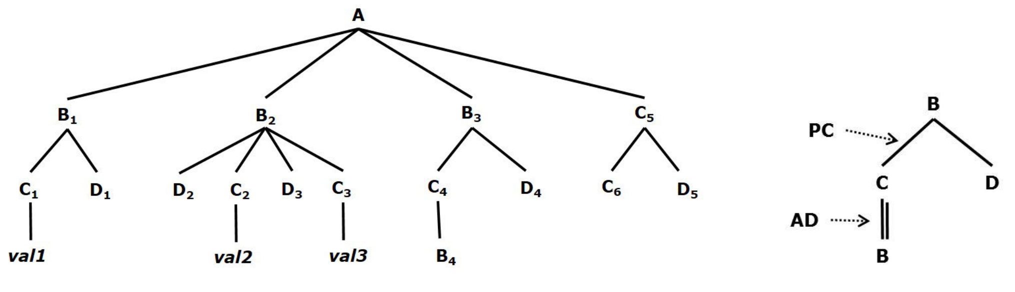
To reduce the cost of a QTP, TPMAs use indexing techniques which partition the XML data-nodes into a number of data-streams. To preserve the structural relationships between data-nodes in different streams, nodes are labeled using one of the many node-labeling schemes. In this section, we will briefly review the main approaches of storage techniques, node-labeling schemes, and TPMAs.

A. MAIN XML DATA STORAGE APPROACHES

The main XML data storage approaches are text file approach, object-oriented approach, relational approach and native approach. Out of these approaches, the native approach is the most efficient and that is why the proposed algorithm is based on this approach [12].

The main indexing techniques in the native approach can be classified as tag-based, tag-level-based, ancestor-based, and ancestor-descendant-based.

In the tag-based approach, nodes having the same name (tag) are put into the same data-stream [16], [17].



(a) Sample XML-tree (b) QTP

FIGURE 1. **A sample XML-tree and a sample QTP.**

For example, in Figure 1a, all the four B nodes are stored in one data-stream and all the six C nodes are stored in another data-stream. Nodes in a stream are sorted in ascending order of their label. Node labeling is discussed in the next subsection.

In the tag-level-based approach, nodes having the same name and level are stored in the same data-stream [18], [19]. For example, in Figure 1a, the three B nodes at Level 1 are stored in one data-stream whereas the B node at level 3 is stored in a different data-stream.

In the ancestor-based approach, nodes having the same name and the same ancestors names are stored in the same data-stream, [20]–[24]. For example, the six C nodes of 1a are stored in three different data-streams. The four C nodes whose ancestors are A and B are stored in the same data-stream, the one C node whose ancestors are A and C are stored in another data-stream, and the one C node whose ancestor is only A is stored in a third data-stream.

In the ancestor-descendant-based approach, nodes with similar ancestors and descendants are stored in the same data-stream [25], [26]. For example, the first two B nodes are stored in the same data-stream and each of the other two B nodes is stored in a separate data-stream.

The proposed storage approach is different from the above mentioned approaches. In this approach, two nodes are stored in the same data-stream if they have the same ancestors and descendants, and their ancestors have the same descendants. Also, in this approach, nodes are labeled using the proposed new node labeling scheme RLP-Scheme, and the nodes with the same label are stored in the same data-stream. This approach creates more data-streams than the other approaches. In general, TPMAs that use storage techniques that partition nodes in more streams are more efficient.

B. XML NODE-LABELING APPROACHES

The above mentioned indexing approaches partition data-nodes into a number of data-streams. To preserve the structural relationships of data-nodes in different streams, each node is systematically labeled. XML node labeling schemes can be classified into three approaches, namely, range-labeling, prefix-labeling, and multiplicative-labeling.

In range-labeling, each data-node n is labeled three numbers, namely, $n.start$, $n.end$, and $n.level$, where $n.start < n.end$ [27]–[29]. If node p is a parent of node c, then $p.start < c.start$, $p.end > c.end$, $p.level = c.level - 1$. The level of the root node is 0. If nodes s_1 and s_2 are siblings, and s_1 is on the left of s_2 in $\tilde{\mathbb{X}}$, then $s_1.end < s_2.start$.

In prefix-labeling, a node at level k is labeled a string of k numbers (self labels) separated by a delimiter. For example, a node n at level 3 is labeled $x.y.z$, where x is the self label of its ancestor node at level 1, y is the self label of its parent node, and z is its own self label. No two siblings are assigned the same self label [30]–[32].

In multiplicative-labeling, each node is labeled a number. The label of a parent node can be computed from that of its child nodes using a function [33]–[35].

The propose node-labeling scheme is a hybrid of multiplicative and prefix labeling approaches, and is explained in Subsection VI.

C. TPMAs

Recent years have witnessed dramatic growth in the amount of research on TPMAs [10]–[12]. Existing TPMAs can be classified in to four main approaches, namely, the relational, the navigational, the sequence-based, and the join-based [10]–[12].

A relational based algorithm uses a relational query processor to evaluate a given QTP [36], [37]. In general, processing QTPs using relational query processor is very costly because of the number of mappings and join operations that are performed. The mappings are: schema mapping, data mapping, QTP mapping, and result mapping. If the XML schema is complex, the schema mapping results in many relational tables. This adds significantly to the cost of the QTP because of the number of join operations that must be performed.

In the navigational approach, TPMAs typically traverse each input XML-tree in preorder and test each node to check if it satisfies the QTP constraints [12], [38]. TPMAs proposed prior to 2002 were all navigational. In general they are less efficient than the join-based algorithms.

Sequence-based algorithms such as PRIX [39] and ViST [40] convert QTPs and XML documents into sequences and then find QTP matches using subsequence matching. However, a set of costly post-processing steps is required to remove false positive candidates and identify the exact matches.

The join-based approach is based on joining data nodes satisfying the structural relationship constraints specified by a given QTP. This approach can be further classified into two approaches, namely, the binary join approach [28], [41]–[43] and the holistic join approach [15], [16], [26], [44]–[46].

In the binary join approach, algorithms find all occurrences of a structural relationship between two nodes. A complex QTP is broken down into a set of basic binary structural relationships, such as, PC and AD relationships between pairs of nodes. They suffer from producing huge intermediate results that don't contribute to the final solution. On the other hand, algorithms using the holistic join approach attempt to evaluate a QTP without breaking it into basic binary structures [19], [23]–[25], [37], [47], [48].

A survey of existing TPMAs can be found in [10]–[12]. Out of the existing TPMA approaches, the holistic approach is the most efficient [12]. The proposed TPMA is a holistic algorithm.

III. XML IN IoT

Currently [49], there are many IoT interoperability proposals supporting XML data format as shown in Table 1. These proposals can be divided into three categories, namely, IoT standard frameworks, projects, and platforms.

IoT generated data in XML format becomes a vital challenge due to the small-sized and yet huge-volume. One approach is proposed by [50]. This approach optimizes storing and accessing XML files in HDFS with the help of a novel unified-indexing service system [51].Using this approach, the performance of service discovering is improved [7]. XML is also used to represent IoT sensor models. This representation is proposed in [52] to allow the description of basic sensor characteristics such as amount of data and its production frequency.

Using indexing functions to ease IoT data processing, authors in [53] represent data using XML format for the reason that XML is becoming the universal standard for data exchanging. In addition, authors provide classification techniques to efficiently handle IoT data. These techniques include classification to enable diverse indexing for service providers and service consumers.

Authors in [54] propose an XML-based scheme to store IoT-generated data in cloud environments. This storage scheme is proposed since XML data model plays a vital role as an intermediate language during the full-fledged transformation to another data model format [55].

A predictive analysis model for service consumers is proposed in [56]. This cloud-enabled IoT-based model utilizes IoT devices as well as cloud computing and XML Web services for faster, secure and reliable data handling. This model enables quick and reliable notification system for abnormality or complications during physical activities [56].

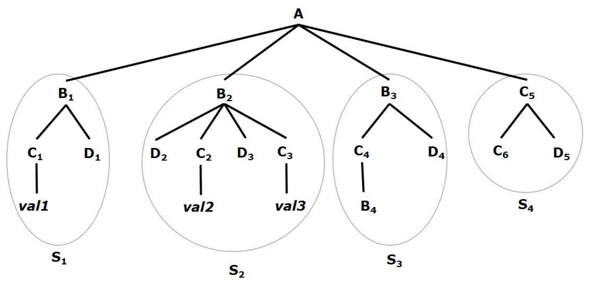


FIGURE 2. S-trees.

Authors in [57] outline a novel architecture for enabling service consumers to construct IoT applications. The configuration settings are stored in XML format and are enabled by cloud-based web applications. Raspberry PI is utilized with Flask MVC to seamlessly use IoT resources represented in XML format.

IV. THE PROPOSED NODE LABELING SCHEME

This section explains the proposed node labeling scheme, RLP-Scheme. But before explaining the scheme, let us define some terms and notations which are used frequently in the rest of this paper.

- **R:** The root node of $\mathbb{X}$.
- R_i**:** The i^{th} child of R, for i = 0, 1, ... $|R|$, where $|R|$ is the count of the children of R.
- **S-tree:** A subtree of $\mathbb{X}$ whose root node is a child of R.
- $S = \{S_1, S2, \ldots, S_{|R|}\}$**:** The set of S-trees in $\mathbb{X}$, where S_i is an S-tree whose root node is R_i. S_i consists of R_i and all its decedents. For example, the XML-tree of Figure 2 consists of four S-trees, namely, S_1, S_2, S_3, and S_4.
- **Labeled-Path:** A sequence, $/tag_0/tag_1/\ldots/tag_n$, from R to node n, where tag_0 is the tag of the root node R [23]. For example, in figure 1(a), /A/B and A/B/C are examples of labeled-paths.
- **RLP:** Root-to-leaf labeled-path. For example, in Figure 1(a), Path /A/B/C is RLP and path /A/B/D is another RLP.
- $P = \{P_1, P_2, \ldots P_{|P|}\}$**:** The set of all distinct RLPs in $\mathbb{X}$, where $|P|$ is the count of distinct RLPs in $\mathbb{X}$.

RLP-Scheme assigns each S-tree, RLP, and node in $\mathbb{X}$ an ID. The ID of S_i is i, where i is an integer number in $\{0, 1, \ldots, |R|\}$. The ID of P_i, ρ_i, is a binary number of size $|P|$ bits and is computed using Equation 1.

$$\rho_i = \mathit{ToBinary}(2^i) \tag{1}$$

where ToBinary() is a function that coverts a decimal number to a binary number. For example, $\mathbb{X}$ of Table 2 has five RLPs, namely, P_1="'/A/B/C"', P_2="'/A/B/D"', P_3="'/A/B/C/B"', P_4="'/A/C/C"' and P_5="'/A/C/D"' and their IDs are, 00001,00010, 00100,01000, and 10000 respectively. Table 2 shows all the RLPs in the XML-tree of Figure 2.

The ID of a node n in S_i has the form $[i, \Phi[n]]$, where $\Phi[n]$ is a sequence of self-labels separated by a delimiter.

TABLE 1. IoT interoperability proposals supporting XML data format.

Category	Name	Website
Frameworks	OGC SWE	www.opengeospatial.org
	ETSI Smart M2M	www.etsi.org/
	AllJoyn	www.openconnectivity.org/developer/reference-implementation/alljoyn
	OIC IoTivity	www.iotivity.org
Platforms	OpenRemote	www.openremote.com/
	Intel IoT Platform	www.maker.pro/intel-iot-platform/projects
	Xively	www.xively.com
	PTC ThingWorx	www.ptc.com/en/products/iiot
	ThingSpeak	www.thingspeak.com/
Projects	Arrowhead	www.arrowhead.eu/
	Ponte	www.eclipse.org/proposals/technology.ponte/
	OpenIoT	www.openiot.eu/
	FIWARE	www.fiware.org/
	SpitFire	www.tugraz.at/en/institutes/iti/research/projects/former-projects/spitfire/

TABLE 2. RLPs.

RLP name	RLP	RLP ID
P_1	/A/B/C	00001
P_2	/A/B/D	00010
P_3	/A/B/C/B	00100
P_4	/A/C/C	01000
P_5	/A/C/D	10000

The self-label of a node is a binary number of size $|P|$ bits, and it is computed from the RLPs that the node belongs to. A leaf-node belongs to one RLP and an inner node can belong to many RLPs. Let SL[n] denote the self-label of node n. If n is the leaf-node of P_i, then SL[n] is computed using Equation 2.

$$SL[n] = \rho_i \tag{2}$$

If n is an inner-node, then its SL[n] is computed from all its RLPs using Equation 3.

$$SL[n] = SL[n_1] \| SL[n_2] \| \ldots \| SL[n_{|n|}] \tag{3}$$

where n_j is the j^{th} child of n, $|n|$ is the count of children of n, and "'$\|$"' is the logical OR operator. The $\Phi[n]$ of any node n is computed using Equation 4.

$$\Phi[n] = \Phi[parent[n]].SL[n] \tag{4}$$

where parent[n] is the parent node of n. The ID of the root node, R, is 0 and the ID of a value node is the same as that of its parent node. Table 3 shows the S-trees, tags, self-labels, Φs, and IDs, of the nodes in the XML-tree of Figure 2.

To minimize the size of bits in a self-label, RLPs can be partitioned into groups. Then, the ID of P_i in Group g can be represented as $[g, \rho_i]$, where g is a group number and is an integer number between 0 and $G - 1$, and G is the number of RLP groups. For example, the five RLPs in $\mathbb{X}$ can be partitioned into two groups based on the tag of R_i. RLPs "'/A/B/C"', "'/A/B/D"', and "'/A/B/C/B"' have the same R_i tag,"'B"', so they belong to the same group, Group 1. Similarly, RLPs "'/A/C/C"' and "'/A/C/D"' have the same R_i tag,"'C"', so they belong to the same group, Group 2. The ID of "'/A/B/C"' will be [1, 001] and that of "'/A/C/D"' will be [2, 10]. For simplicity, in this paper we will assume that all the RLPs are in one group,and thus, the group number can be ignored. Let us now define "'identical S-trees" and "'identical nodes"' terms which are used frequently in the rest of this paper.

TABLE 3. Trees, tags, self-labels, Φs, and IDs, of the nodes in the XML-tree of Figure 2.

S-tree	Tag	Self-label	Φ	ID
S_1	B_1	00011	00011	[1, 00011]
	C_1	00001	00011.00001	[1, 00011.00001]
	D_1	00010	00011.00010	[1, 00011.00010]
S_2	B_2	00011	00011	[2, 00011]
	D_2	00010	00011.00010	[2, 00011.00010]
	C_2	00001	00011.00001	[2, 00011.00001]
	D_3	00010	00011.00010	[2, 00011.00010]
	C_3	00001	00011.00001	[2, 00011.00001]
S_3	B_3	00110	00110	[3, 00110]
	C_4	00100	00110.00100	[3, 00110.00100]
	B_4	00100	00110.00100.00100	[3, 00110.00100.00100]
	D_4	00010	00110.00010	[3, 00110.00010]
S_4	C_5	11000	11000	[4, 11000]
	C_6	01000	11000.01000	[4, 11000.01000]
	D_5	10000	11000.10000	[4, 11000.10000]

Definition 1 (Identical nodes): Nodes n and m are identical if $\Phi[n] = \Phi[m]$.

Definition 2 (Identical S-trees): S-trees S_i and S_j are identical, if for every node in S_i there is an identical node in S_j and for every node in S_j there is an identical node in S_i.

RLP-Scheme has two advantageous properties that no other node labeling scheme has. The first advantage is that, from the ID of a node, the RLPs of the node and that of its ancestor nodes can be computed. This property can speedup query processing because it can minimize the number of nodes accessed. The second advantage of RLP-Scheme is that it saves a lot of storage space and CPU time when storing and processing an XML-tree with many identical S-trees. If an XML-tree has many identical S-trees, and if each of these S-trees has no identical nodes, then it is enough to store only the IDs of the these S-trees, and the Φ values of the nodes of only one of these S-trees. The storage of S-trees with identical nodes will be explained in the next section. For example, in DBLP dataset, one S-tree has more than 80,000 identical S-trees. If each of these S-trees has 50 non-identical nodes, then, it is enough to store 50+80,000=80,050 IDs instead of 50∗80,000=4,000,000 IDs. Also during query processing, a given query is only matched against one of these 80,000 identical S-trees, which minimizes query cost significantly as explained in subsequent sections.

V. THE PROPOSED STORAGE STRUCTURE

XML-trees can contain many repeated subtree structures. We can take advantage of these repeated subtree structures to minimize storage space and speedup query processing time.

To evaluate XML queries efficiently, two new algorithms and a new node labeling scheme are proposed in this work. The first algorithm, *RLP-Index*, takes as input $\mathbb{X}$ and generates an index of $\mathbb{X}$ called $\tilde{\mathbb{X}}$. The second algorithm, *MatchQTP*, takes as input $\tilde{\mathbb{X}}$ and a QTP, and gives as output the solution of the QTP, $\|QTP\|$. The proposed node labeling scheme, *RLP-Scheme*, assigns nodes of identical subtree structures the same label. It is used by RLP-Index to identify identical subtree structures. But before explaining the proposed algorithms and the proposed node labeling scheme in detail, let us define some terms and notations that are used in the rest of this paper.

- $R_1, R_2, \ldots, R_{|R|}$**:** The children of R where $|R|$ is their count. For example, B_1, B_2, B_3 and C_5 of Figure 2.
- **Identical siblings:** Sibling nodes with identical subtree structures. Nodes n_1 and n_2 are identical siblings if the children of n_1 have the same name as the children of n_2, and it is also recursively true for all their decedents.
- **Dtree:** A subtree of $\mathbb{X}$ rooted at R_i. A Dtree rooted at R_i contains all the decedents of R_i in $\mathbb{X}$. For example, the subtrees S_1, S_2, S_3, and S_4 of Figure 2 are Dtrees.
- **Ttree (Template tree):** A subtree created from a Dtree by merging all the identical siblings in the Dtree. For example, in the XML-tree of Figure 1a, there are four Dtrees and three Ttrees.
- **Dnode, Tnode:** A Dtree node and a Ttree node respectively.
- **Qnode, Qroot, Qtarget:** A QTP node, a QTP root node, and a QTP output (target) node respectively. For example, in QTP "'//A[//B]/C"', A, B, and C are Qnodes; A is the Qroot; and C is the Qtarget.

A. THE PROPOSED INDEX STRUCTURE $\tilde{\mathbb{X}}$

The proposed index, $\tilde{\mathbb{X}}$, has five main components. Its first component is a DataGuide, DG. In $\tilde{\mathbb{X}}$, each DG node (DGnode) is assigned a unique label, DGnodeLabel. A DGnodeLabel consists of four numbers and is of the form [DGnodeNbr, GID, DGnodeSelfLabel, DGnodeID]. A DGnodeNbr is a unique integer number from the set 1, 2, …, $|DG|$, where $|DG|$ is the count of DGnodes in DG; GID is a group ID; DGnodeSelfLabel is a self-label and is computed as explained in Subsection VI; and DGnodeID is a Dewey ID and is used to find QTP matches in DG. Figure 4 shows a DG of the XML-tree of Figure 1a. To speed up query processing, DG is partitioned into ancestor-based streams. DGnodes with the same name and the same ancestor names are put into the same stream, DGnode-stream. The nodes in a DGnode-stream are sorted in ascending order of their DGnodeNbr.

The second $\tilde{\mathbb{X}}$ component is the T-Tnode table. It has two columns, namely, DGNodeNbr, and Tnode-stream. The number of rows in the T-Tnode table is equal to the count of DGnodes, $|DG|$. It maps each DGnode to a Tnode-stream. A Tnode-stream is a sorted list of TnodeIDs. A TnodeID is of the form [TID, p.s], where TID is the ID of the corresponding Ttree and p and s are as explained in SubsectionVI. TID is a unique integer number that identifies a Ttree. Table 4(a) is an example of T-Tnode table populated from the sample XML-tree of Figure 1a.

The third component of $\tilde{\mathbb{X}}$ is a table called *T-Dtree*. It has three columns, namely, *TID*, *TStructure*, and *Dtree-stream*. For each Ttree, we store its ID in *TID*, its structure in *TStructure*, and the IDs of its corresponding Dtrees in *Dtree-stream*. Table 4(b) is as example of a T-Dtree. Each row in this table contains a Ttree and its corresponding sorted list of Dtrees, a Dtree-stream. The number of Dtree-streams is equal to the number of Ttrees. After merging all the identical-sibling Dnodes in each Dtree of a Dtree-stream, the Dtrees in that stream become structurally identical to each other and to the Ttree which is in the same row.

Each Tnode in a T-Tnode table physically points to a Dtree-stream but logically it points to a Dnode-stream. A Dtree-stream is physically one stream but it can be mapped to n Dnode-streams, where n is the number of nodes in the corresponding Ttree. A DnodeID is of the form [DID, p.s]; and a TnodeID is of the form [TID, p.s]. The value of the second component of each DnodeID ($p.s$), is equal to that of its corresponding TnodeID. For example, in Figure 3, Ttree [1] corresponds to Dtrees [1] and [2]; and thus, the second component of TnodeID [1,3.2] is equal to that of DnodeIDs [1,3.2] and [2,3.2]. Since a DnodeID can be computed from a TnodeID, there is no need to store in $\tilde{\mathbb{X}}$ a Dnode who has no identical siblings.

The fourth $\tilde{\mathbb{X}}$ main component is called *T-identicalSiblings* table. It contains all the identical sibling data-nodes in $\mathbb{X}$ which are at levels 2 or more. It has two columns, namely *TnodeID* and *Siblings-stream*. Each element of a Siblings-stream is of the form [DID, count] and represents siblings' Dtree and count. The DnodeID of each sibling Dnode in $\mathbb{X}$ can be computed from T-identicalSiblings. A DnodeID is of the form [DID, p.s, order], where $p.s$ is

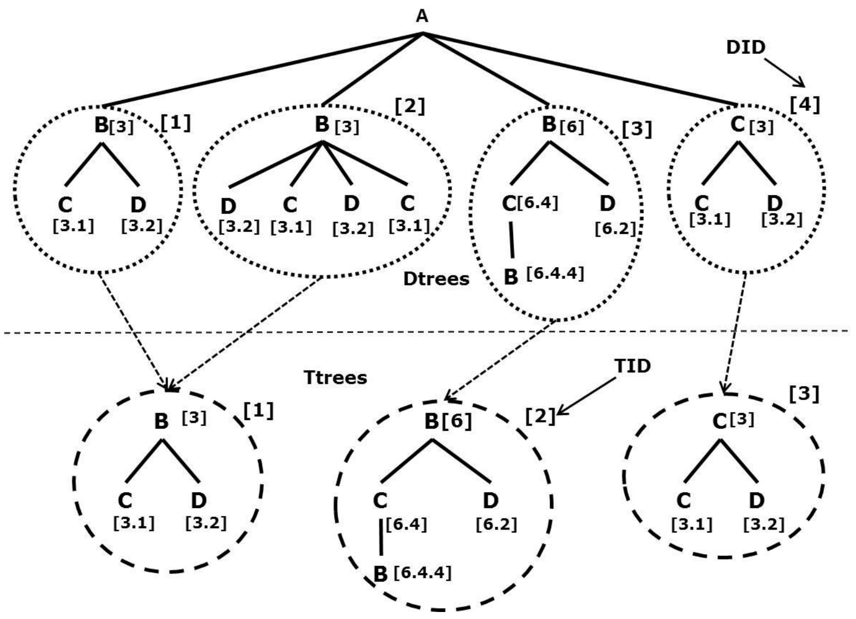


FIGURE 3. The Dtrees and Ttrees of the sample XML-tree of Figure 1a.

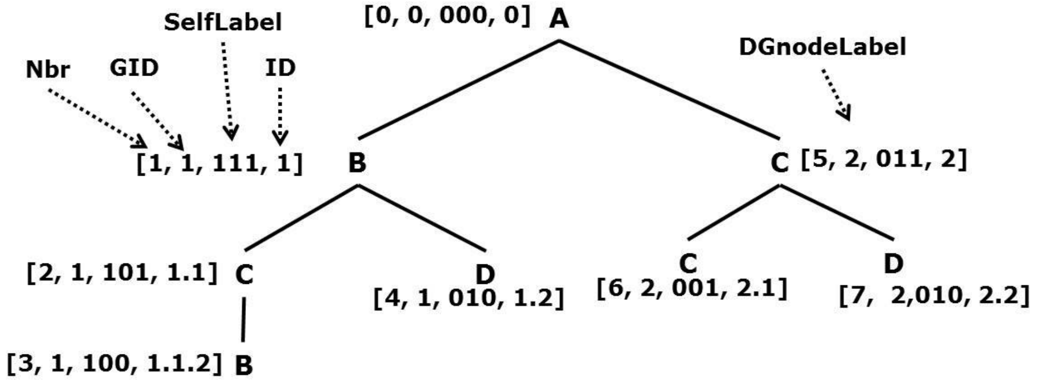


FIGURE 4. The DataGuide (DG) of the XML-tree of Figure 1a.

identical to that of the corresponding TnodeID, and *order* can be computed from *count*. Table 4(c) is an example of T-identicalSiblings populated from the sample XML-tree of Figure 1a.

The last $\tilde{\mathbb{X}}$ table is the values table and it contains all the values in $\mathbb{X}$, Table 4(d). It has three fields, namely *Value TID*, and *DnodeID*. This table is horizontally partitioned. Each partition belongs to an RLP whose corresponding data nodes have values. Values which correspond to the same RLP are stored in the same partition. Each partition has a name of the form T-Value-RLPid. For example, the values that correspond to RLPid [1,2] are stored in a partition called "T-Value-1-2".

B. THE PROPOSED $\tilde{\mathbb{X}}$ BUILDER ALGORITHM, RLP-Index

RLP-Index takes as input $\mathbb{X}$ and gives as output $\tilde{\mathbb{X}}$ and is depicted in Algorithm 1. The algorithm first initializes all the components of $\tilde{\mathbb{X}}$ at Line 1. It then reads the first Dtree from $\mathbb{X}$ and assigns it a DID, Line 2. At Line 4, RLP-Index creates a Ttree out of the new Dtree and labels its Tnodes using RLP-Scheme. If a Ttree identical to the new Ttree is not already in the T-Dtree table, RLP-Index labels the new Ttree, adds any new of its labeled paths into DG, and adds its Tnodes into the T-Tnode table, lines 6 to 10. At Line 11, RLP-Index adds the new Dtree to the T-Dtree table. If the new Dtree has identical sibling nodes, then RLP-Index adds their Dtree and count to the T-identicalSiblings table, Line 12. Any values in the new Dtree are added into the T-values table at Line 13. At Line 14, it reads the next Dtree and repeats lines 3 to 15. The algorithm terminates when all the Dtrees in $\mathbb{X}$ are processed.

For example, given the XML-tree of Figure 3, RLP-index first reads the first Dtree, which is the subtree in the first dotted circle, and assigns it a DID of 1. It then maps this Dtree to a Ttree by merging any of its identical siblings. In Figure 3, the dotted arrow connects each Dtree to its corresponding

TABLE 4. T-Tnode, T-Dtree, T-IdenticalSiblings and T-value-1-1 of $\tilde{\mathbb{X}}$.

DGNodeNbr	Tnode-stream
1	[1, 3] [2, 6]
2	[1, 3.1] [2, 6.4]
3	[2, 6.4.4]
4	[1, 3.2] [2, 6.2]
5	[3, 3]
6	[3, 3.1]
7	[3, 3.2]

(a) T-Tnode

TID	TStructure	Dtree-Stream
1	A[3], B[3.1], C[3.2]	1, 2
2	B[6][6.4.4], C[6.4], D[6.2]	3
3	C[3][3.1], D[3.2]	4

(b) T-Dtree

TnodeID	Siblings-Stream
[1, 3.1]	[2, 2]
[1, 3.2]	[2, 2]

(c) T-IdenticalSiblings

Value	TID	DnodeID
val1	1	[1,3.1,1]
val2	1	[2,3.1,1]
val3	1	[3.3.1,2]

(d) T-Value-1-1

```
Algorithm 1 RLP-Index
Require: $\mathbb{X}$
Ensure: $\tilde{\mathbb{X}}$
 1: Initialize($\tilde{\mathbb{X}}$)
 2: DID ← ReadDtree($\mathbb{X}$)
 3: while DID do
 4:     $T$ ← MergeIdenticalSiblings(DID)
 5:     TID ← GetTID(T-Dtree,T)
 6:     if ! TID then
 7:         TID ← GenerateLabel(Ttree, T)
 8:         DGnodes ← Add(DG, TID)
 9:         Add(T-Tnode, DG, TID)
10:     end if
11:     Add(T-Dtree, TID, DID)
12:     Add(T-identicalSiblings, TID, DID)
13:     Add(T-values, TID, DID)
14:     DID ← ReadDtree($\mathbb{X}$)
15: end while
16: return $\tilde{\mathbb{X}}$
```

Ttree. Since this is first Ttree created it is assigned a TID of 1. The Ttree nodes are labeled as shown in the figure and stored in the T-Dtree table. Since the Dtree has no siblings, none of its nodes are stored. But the label of the Dtree, which is also 1, is stored in Table 4(b). After it finishes processing the first Dtree, the algorithms reads the second Dtree, which is labeled as [2] in the figure. Since this Dtree has some identical siblings, it will merge the siblings and then convert the Dtree to Ttree. The Ttree of this Dtree is identical to that of the first Dtree and thus there is no need to store its Ttree. It will only store its label, which is 2. It will also stores the labels of the siblings in Table 4(c). The algorithm will read the remaining Dtrees one by one and convert them to Ttrees and store them in the same way. Table 4 shows all the values that are stored at the end of the Algorithm.

C. THE PROPOSED TPMA, MatchQTP

MatchQTP takes as input $\tilde{\mathbb{X}}$ and a QTP and gives the solution of the QTP, $\|QTP\|$, as output. MatchQTPconsists of two main functions, namely, MatchQTPValues and MatchQTPStructure. MatchQTPis depicted in Algorithm 2. If a QTP has value predicates, MatchQTP call the function MatchQTPValues otherwise its calls MatchQTPStructure.

```
Algorithm 2 MatchQTP
Require: $\tilde{\mathbb{X}}$, QTP
Ensure: $\|QTP\|$
1: if HasValuePredicates(QTP) then
2:     MatchQTPValues($\tilde{\mathbb{X}}$, QTP)
3: else
4:     MatchQTPStructure($\tilde{\mathbb{X}}$, QTP)
5: end if
6: return
```

The function MatchQTPValues is depicted in Algorithm 3. It takes as input a QTP and $\tilde{\mathbb{X}}$ and gives as output $\|QTP\|$. At line 1, the algorithm extracts each predicate value (Qvalue) in the QTP. It then identifies the DGNodeNbr of each Qvalue, Line 2. It then searches for the first Qvalue in its corresponding T-value, Line 3. If a match is found, it checks if the corresponding T-tree matches the QTP, Line 5. If they match, then MatchQTPValues checks if the corresponding D-tree contains the other Qvalues, Line 7. If the D-tree contains all the Qvalues, then it is sent to the output, Line 8. At line 11, MatchQTPValues searches for the next Qvalue. It loops between lines 4 and 12 until it finds all the QTP matches.

The order MatchQTPStructure moves from one component of $\tilde{\mathbb{X}}$ to another is depicted in Figure 5. Given a QTP, MatchQTPStructure first searches DG for a DGTargetNode, a DGnode that matches a Qtarget. The search then moves to the target-Tnode-stream, which is a Tnode-stream in the T-Tnode

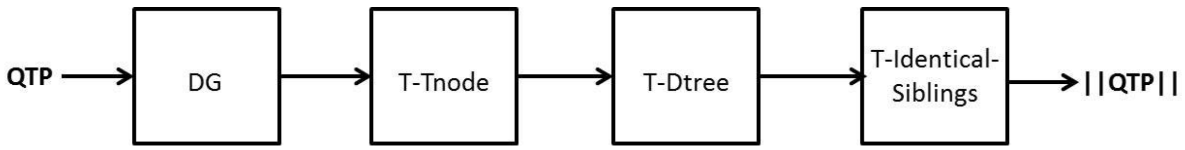


FIGURE 5. The order MatchQTP moves between the main components of $\tilde{\mathbb{X}}$.

Algorithm 3 MatchQTPValues

```
Require: $\tilde{\mathbb{X}}$, QTP
Ensure: $\|QTP\|$
 1: QValue ← GetQTPValues(QTP)
 2: QValDGNbr ← GetQValuesDGNbrs(QTP)
 3: Record ← GetQValueMatch(QValue[1], QValDGNbr[1])
 4: while Record do
 5:     DID ← MatchQTree(Record)
 6:     if DID then
 7:         if MatchAllQValues(DID, QValue, QValueDGNbr) then
 8:             Output(DID, QTP)
 9:         end if
10:     end if
11:     Record ← GetNextQValueMatch(QValue[1], QValDGNbr[1])
12: end while
13: return
```

table that corresponds to the DGTargetNode. As mentioned above, each DGnode maps to one Tnode-stream, and each Tnode maps to one Dnode-stream and one Siblings-stream. For each Tnode, in the target-Tnode-stream, that matches the QTP, the algorithm sends its corresponding Dnode-stream and its corresponding sibling nodes to the output. MatchQTP then moves to the next DGTargetNode and repeats the above steps again. It terminates when it processes all the DGTargetNodes.

MatchQTP is depicted in Algorithm 2. At line 2, it finds the stream of DGtargetNode in DG. It then selects the first DGtargetNode, Line 3. Let us refer to this DGtargetNode as the current-DGtargetNode. It then finds all DG subtrees that match the given QTP and contain the current-DGtargetNode, Line 5. Let these DG subtrees be denoted as Φ_i, for $i = 1, 2, \ldots, |\Phi|$, where $|\Phi|$ is their count. Also, let $\hat{\Phi}_i$ denote the root node of Φ_i; $\Phi_{i,j}$, for $j = 1, 2, \ldots, |\Phi_i|$, denote the nodes of Φ_i; $\beta[\Phi_{i,j}]$ denote the self-label of $\Phi_{i,j}$. The self-label of the root-node of Φ_i, $\hat{\Phi}_i$, is computed using Equation 5.

$$\beta[\hat{\Phi}_i] = \beta[\check{\Phi}_{i,1}]\|\beta[\check{\Phi}_{i,2}]\| \ldots \|\beta[\check{\Phi}_{i,|\check{\Phi}_i|}] \tag{5}$$

where $\check{\Phi}_{i,j}$ is the j^{th} leaf node of Φ_i and $|\check{\Phi}_i|$ is the number of leaf nodes in Φ_i. At lines 6 to 15, MatchQTP computes the self-labels of each Φ_i using Equation 5. The search then moves to the Tnode-stream, in the T-Tnode table, that corresponds to the current-DGtargetNode, lines 16 to 25. At Line 19, MatchQTP calls the boolean function, *MatchTnode* to check if the current Tnode belongs to a Ttree that matches the QTP. *MatchTnode* will be discussed shortly. If *MatchTnode* return true, then a partial output of the QTP is returned at lines 20 and 21. At Line 23, the search for QTP matches moves to the next Tnode. When all the Tnodes that correspond to the current-DGtargetNode are processed, MatchQTP moves to the next DGtargetNode at Line 26. It repeats lines 4 to 27 many times until it returns the complete solution of the given QTP.

The function *MatchTnode* is depicted by Algorithm 5. It checks if a Tnode belongs to a Ttree that matches a given QTP. It takes as input a Tnode, Φ_i, and $\hat{\Phi}_i$. At Line 4, the algorithm computes the root level of the current Φ_i. At Line 5 it extracts the ancestor of the input Tnode which corresponds to the Qroot; and at Line 6 its extracts the ancestors self-label. At lines 7 and 8 it uses Equation 6 to check if the self-label of the input Tnode matches $\beta[\hat{\Phi}_i]$. If they match, it calls the function *MatchTree* to check if the corresponding Ttree matches the QTP, Line 9. *MatchTree* is a boolean function and it takes as input a Tnode and a QTP. It then searches in the T-Dtree for a Ttree of the Tnode. It returns true if the Ttree matches the QTP otherwise it returns false. If the current Φ_i doesn't match the QTP, it goes back to Line 3 to check the next one.

$$M = (\beta[\hat{n}]\&\beta[\hat{\Phi}_1])\| \ldots \|(\beta[\hat{n}]\&\beta[\hat{\Phi}_{|\phi|}]) \tag{6}$$

For example, given the Dtrees of Figure 3 and the query /A/B/C/B, the MatchQTP first searches the DataGuide in Figure 4, for a path that matches the query. It then retrieves the Nbr of the last node in the path which is 3. Next the algorithm searches the table T-Tnode, shown in Table 4 (a), for a row with DGNodeNbr of 3. The data-stream in that row is [2, 6.4.4]. It then searches the T-Dtree, in Table 4 (b), for a row with TID 2 and retrieves all the DIDs which are in the corresponding D-tree-stream column. In this case we only have one DID which is 3. At last, the algorithm will return the output of the query as 3.6.4.4 by concatenating the DID that it retrieved from Table 4(b) and the value 6.4.4 that it retrieved from Table 4(a).

VI. THE PROPOSED NODE LABELING SCHEME, RLP-SCHEME

In RLP-Scheme, node labels are computed from RLPs. In this scheme, RLPs in $\mathbb{X}$ are partitioned into a number of groups. RLPs with the same R_i node names are put into the same group. Depending on the application and the characteristics

Algorithm 4 MatchQTPStructure

```
Require: $\tilde{\mathbb{X}}$, QTP
Ensure: $\|QTP\|$
 1: $\|QTP\| \leftarrow$ NULL
 2: DGTargetStreamCursor $\leftarrow$ GetDGTargetStream(DG, QTP)
 3: DGTargetNbr $\leftarrow$ GetDGTargetNumber(DG, DGTargetStreamCursor)
 4: while DGTargetNbr do
 5:   $\Phi \leftarrow$ GetMatchingDGSubtrees(DG, DGTargetNbr, QTP)
 6:   if $\Phi$ then
 7:     for i = 1 to $|\Phi|$ do
 8:       $\hat{\Phi}_i \leftarrow$ GetRootNode($\Phi_i$)
 9:       $\beta[\hat{\Phi}_i] \leftarrow$ NULL;
10:       for j = 1 to $|\Phi_i|$ do
11:         if IsLeafNode($\Phi_{i,j}$) then
12:           $\beta[\hat{\Phi}_i] \leftarrow \beta[\hat{\Phi}_i] \| \beta[\Phi_{i,j}]$
13:         end if
14:       end for
15:     end for
16:     TnodeStreamCursor $\leftarrow$ GetTnodeStream(T-Tnode, DGTargetNbr)
17:     TargetTnode $\leftarrow$ GetTargetTnode(T-Tnode, TnodeStreamCursor)
18:     while TargetTnode do
19:       if MatchTnode(TargetTnode, $\Phi$, $\hat{\Phi}$, QTP) then
20:         $\|QTP\| \leftarrow \|QTP\|$ + Output(T-Dtree, TargetTnode)
21:         $\|QTP\| \leftarrow \|QTP\|$ + Output(T-identicalSiblings, TargetTnode)
22:       end if
23:       TargetTnode $\leftarrow$ GetTargetTnode(T-Tnode, TnodeStreamCursor++)
24:     end while
25:   end if
26:   DGTargetNbr $\leftarrow$ GetDGTargetNumber(DG, DGTargetStreamCursor++)
27: end while
28: return $\|QTP\|$
```

Algorithm 5 MatchTnode

```
Require: CurrentTnode, $\Phi$, $\hat{\Phi}$, QTP
Ensure: MatchFound
 1: MatchFound $\leftarrow$ FALSE
 2: i $\leftarrow$ 1
 3: while i <= $|\Phi|$ && NOT MatchFound do
 4:   QrootLevel $\leftarrow$ GetRootLevel($\Phi_i$)
 5:   k $\leftarrow$ GetAncestor(CurrentTnode, QrootLevel)
 6:   n $\leftarrow$ GetSelfLabel(k)
 7:   m $\leftarrow$ n && $\beta[\hat{\Phi}_i]$
 8:   if m == $\beta[\hat{\Phi}_i]$ then
 9:     MatchFound $\leftarrow$ MatchTree(CurrentTnode, QTP)
10:   end if
11:   i++
12: end while
13: return MatchFound
```

of $\mathbb{X}$, other conditions of partitioning RLPs can also be used. Each group is then uniquely labeled an integer number from the set $\{1, 2, 3, \ldots, |G|\}$ where $|G|$ is the number of distinct R_i node names in $\mathbb{X}$. For example, in $\mathbb{X}$ of Figure 1a, there are two RLP groups because there are two distinct R_i node names, namely B and C. RLPs "'/A/B/C'", "'/A/B/D'", and "'/A/B/C/B'" belong to Group 1 whereas RLPs "'/A/C/C'" and "'/A/C/D'" belong to Group 2.

Each RLP is assigned a unique ID, **RLPid**. An RLPid is of the form [g,s], where g is a group number and s is a self-label. A self-label of an RLP in group g is a unique binary number of size $|g|$ bits, where $|g|$ is the number of distinct RLPs in group g. For each self-label, one of the g bits is set to 1 and the rest are set to 0. For example, in Figure 1a, the self-labels of the three RLPs in Group 1 are 001, 010, and 100. The advantage of putting RLPs into different groups is to minimize the number of bits needed by the self-labels.

In RLP-Scheme, a node is assigned an ID, a *nodeID*, of the form [g,p.s] where g is its group number, p is the self-label of all its ancestors, and s is its self-label. A leaf node belongs to one RLP whereas an inner node may belong to more than one RLPs. The self-label of a leaf node is equal to the self label of its RLP. The self-label of an inner node n is the sum of the self-labels of its distinct RLPs and can be computed using Equation 7.

$$\beta[n] = \beta[n_1] \| \beta[n_2] \| \ldots \| \beta[n_{|n|}] \quad (7)$$

where $\beta[n]$ is the self-label of node n, n_i is the i^{th} child of n, $|n|$ is the count of the children of n, and $\|$ is the logical OR operator. For example, Figure 2 shows nodes labeled using RLP-Scheme. From the self-label of a node, all the RLPs of a node can be identified. Since the nodeID of a node includes the self-labels of its ancestors, from the nodeID of a node, the RLPs of each of its ancestors can also be identified.

RLP-Scheme is depicted in Algorithm 6. It takes as input a Dtree and gives as output a labeled Dtree. At Line 1, it initializes a DataGuide [58], a stack, a string variable called RLP, and an array called Dnode. At Line 3, it assigns the i^{th} node of Dtree to Dnode. The RLP of the new Dnode element is updated at Line 4. If the new Dnode element is a leaf node, and the new RLP is not in the DataGuide, it labels the new RLP and stores it in the DataGuide, Line 6. The new Dnode element is then labeled with the label of its corresponding RLP. At Line 7, the parent of the new Dnode element is popped from the stack. The label of the parent is updated by ORing it with the label of the new Dnode element, Line 8. Then the parent is pushed back to the stack, Line 9. If the new Dnode element is an inner node, then its label is initialized to a 0 and it is pushed into the stack. If there is any node in the stack whose level is greater or equal to the level of the new Dnode element, then the node is popped from the stack and its label is ORed to the label of its parent, lines 11 and 17. At Line 18, RLP-Scheme prefixes the label of each Dnode element with the label of its ancestors.

Algorithm 6 RLP-Scheme

```
Require: Dtree
Ensure: Labeled Dtree
 1: Init(DG, Stack, RLP, Dnode, Level)
 2: for i = 1 to |Dtree| do
 3:    Dnode[i] ← Dtree.GetNode(Dtree, i)
 4:    UpdateRLP(Dnode[i])
 5:    if IsLeaf(Dnode[i]) then
 6:       $\beta[Dnode[i]] \leftarrow$ UpdateDG(RLP)
 7:       ParentNode ← Pop()
 8:       $\beta[ParentNode] \leftarrow \beta[ParentNode] \,||\, \beta[Dnode[i]]$
 9:       Push(ParentNode)
10:    else
11:       ClearStck(Level, $\beta[Dnode[i]]$)
12:       $\beta[Dnode[i]] \leftarrow 0$
13:       Push(Dnode[i])
14:    end if
15:    Level ← GetLevel(Dnode[i])
16: end for
17: ClearStack(Level, 0)
18: AddAncestorLabels(Dnode)
19: return Labeled Dtree
```

VII. PERFORMANCE ANALYSIS

Many experiments were conducted to study the performance and scalability of the proposed algorithm, MatchQTP. It was compared with four state-of-the-art TPMAs, namely TwigStack [16], TwigList [59], TJFast [45] and CIS-X [21]. But before we discuss the experimental results, let us go through the experimental setup.

A. EXPERIMENTAL SETUP

All the experiments were conducted on Windows 7 system with a 2.5 GHz Intel Core2 CPU, 4 GB RAM. We used C# to implement the four TPMAs as well as the proposed algorithm. To evaluate the performance of the proposed algorithm, the parameter "'query execution time"' was used. Execution time of a query is the sum of the CPU and I/O costs of the query. Each query execution time reported in this paper is the average of 10 runs. Before each run, the memory was flashed to get rid of cached pages.

1) EXPERIMENTAL DATASETS

The experiments were done using four datasets, namely DBLP [60], [61], XMark [62], TreeBank [61], and DONS [63]. The DBLP and TreeBank datasets are real world benchmark datasets and XMark is a benchmark synthetic dataset. We chose these datasets since they cover a wide range of structural characteristics used in the literature. DONS is a combination of real and synthetic dataset. It contains more identical siblings than the above three datasets. The real data in the DONS dataset was taken from the Disease Outbreak Notification System (DONS) database application that was developed in King Fahd University of Petroleum and Minerals (KFUPM) [63]–[65]. The DONS application was first developed as a relational database application. It is now in the process of developing it as an XML database application. Table 5 shows some characteristics of the experimental datasets.

TABLE 5. Some statistics of the experimental datasets.

Dataset	Size (MB)	Elements (Millions)	Distinct Elements	Max Depth
DBLP	127	3.3	35	6
XMark	112	1.7	74	12
TreeBank	84	2.4	250	36
DONS	123	3.2	64	5

TABLE 6. Experimental queries.

Query	Dataset	XPath Expression
D1	DBLP	// dblp/inproceedings/booktitle
D2	DBLP	// dblp/inproceedings[/title]/author
D3	DBLP	// dblp/article[/author][//title]//year
D4	DBLP	// dblp/inproceedings[//cite/label][/title]/author
D5	DBLP	// dblp/article[/author][//title][//url][//ee]//year
D6	DBLP	//article[//mdate][//volume][//cite//label]//journal
X1	XMark	/site/regions//item/location
X2	XMark	/site/closed_auctions/closed_auction/price
X3	XMark	//open_auction[/current][//annotation//text]//quantity
X4	XMark	//closed_auction[/seller][//itemref][//bold]/date
X5	XMark	//item[location][//mailbox/mail//emph]/description //keyword
X6	XMark	//people//person[//address/zipcode][/id]/profile [//age]/education
T1	TreeBank	//S/VP//PP[/NP/CD]/IN
T2	TreeBank	//EMPYT[/_PERIOD_]//S[//PP//NN][//VBD]//NP
T3	TreeBank	//S[/NP/NN][//VBD][//TO]/VP/PP[/IN]//CD
T4	TreeBank	//EMPTY/S//NP[/SBAR/WHNP/PP//NN]/_COMMA_
T5	TreeBank	//PP//NP[//VP[//VBG] [//SBAR//NNPS]]//_COMMA_
T6	TreeBank	//_QUOTES_//S[//VP/SBAR[/_NONE_][//SQ/MD] [/S/NP]]//_COMMA_
C1	DONS	//case/HU_city/province_name
C2	DONS	//case//disease_name//symptoms
C3	DONS	//case//disease[/lab//labDiagnosisvalueDB] //symptom_EN_name
C4	DONS	// healthunit/[/HU_EN_name][//province_X_coordinates] [//province_Y_ccordinates]/HU_city
C5	DONS	//disease_name[/labdiagnoses/labDiagnosisvalueDB] //labspeciesvalue
C6	DONS	/cases[//case/ID][//nat][//sex][//age][//city]//occupation

2) EXPERIMENTAL QUERIES

Many of the queries used in this work were selected from those in [21], [22], [59]. Table 6 shows all the experimental queries.

B. EXPERIMENTAL RESULTS AND ANALYSIS

1) CONSTRUCTION TIME OF $\tilde{\mathbb{X}}$

Table 7 shows the construction times of $\tilde{\mathbb{X}}$. In the conducted experiments, the maximum construction time was only 89 seconds, which implies that the proposed algorithms are practical. The construction time of $\tilde{\mathbb{X}}$ is low because $\mathbb{X}$ is scanned only once. The construction time also depends on the size of $\mathbb{X}$ and the number of Ttrees in $\tilde{\mathbb{X}}$. If the number of Ttrees in $\tilde{\mathbb{X}}$ is low then the construction time is higher. This

TABLE 7. $\tilde{\mathbb{X}}$ **construction time (sec).**

Dataset	DBLP	XMark	TreeBank	DONS
Time	75	31	22	89

is because the Table T-TDtree is implemented as a hash table. Each time RLP-Index reads a new Dtree from $\mathbb{X}$, it must store it in the hash table. If the new Dtree hashes to an existing Ttree, then this may result in collision, and thus the structures of the Dtree and the corresponding Ttree must be compared. But if the Dtree doesn't hash into any of the existing Ttrees, then no collision resolution is needed and the Dtree is stored directly in to the hash table. As can be seen from the Table 5, the sizes of XMark and DONS are nearly the same; but $\tilde{\mathbb{X}}$ construction time of DONS is higher than that of XMark. This is because DONS results in more collisions and has many identical siblings than XMark.

2) PERFORMANCE ANALYSIS OF MatchQTP

MatchQTP was compared with TwigStack, TwigList, TJFast, and CIS-X. The experimental results of the comparison is shown in Figures 6 to 10. Since the range of processing time was very large, a logarithmic scale was chosen for the figures.

MatchQTP vs. TwigStack: As can be seen from Figure 6, MatchQTP outperformed TwigStack by a factor of 26 on the average. TwigStack uses tag-based streams and thus it creates few number of data-streams. Each stream, on the average, contains many data nodes. If a QTP contains n distinct node names, then TwigStack reads the corresponding n streams. Many of the nodes it reads don't contribute to the final answer. It performs a very expensive containment join operation between nodes of different streams. It also generates many intermediate results and performs path merging. MatchQTP reads only data nodes that appear in the final output. For example, in processing D1 Twigstack read 12 times more data nodes than the proposed algorithm. MatchQTP doesn't produce intermediate results, it doesn't perform any merge operation between data nodes or paths, and it only reads the data-streams that belong to the target node.

MatchQTP vs. TwigList: TwigList, like TwigStack, uses tag-based streams and thus both of them have the same number of data-streams. To process a QTP of n distinct nodes, it reads n data-streams. Unlike TwigStack it doesn't suffer from huge intermediate results but it must perform many containment joins. It also processes many data nodes which don't appear in the final solution of a given query. As shown by Figure 7, MatchQTP outperforms TwigList by a factor of 22.

MatchQTP vs. TJFast: Like TwigStack and TwigList, TJFast also uses tag-based streams; but it uses different node labeling scheme called extended Dewey [45] to reduce the number of data-streams it reads. If a node n is labeled using extended Dewey, then from the label of node n, the labels and the names of its ancestors can be extracted. This enables TJFast to process any QTP by reading only the data-streams

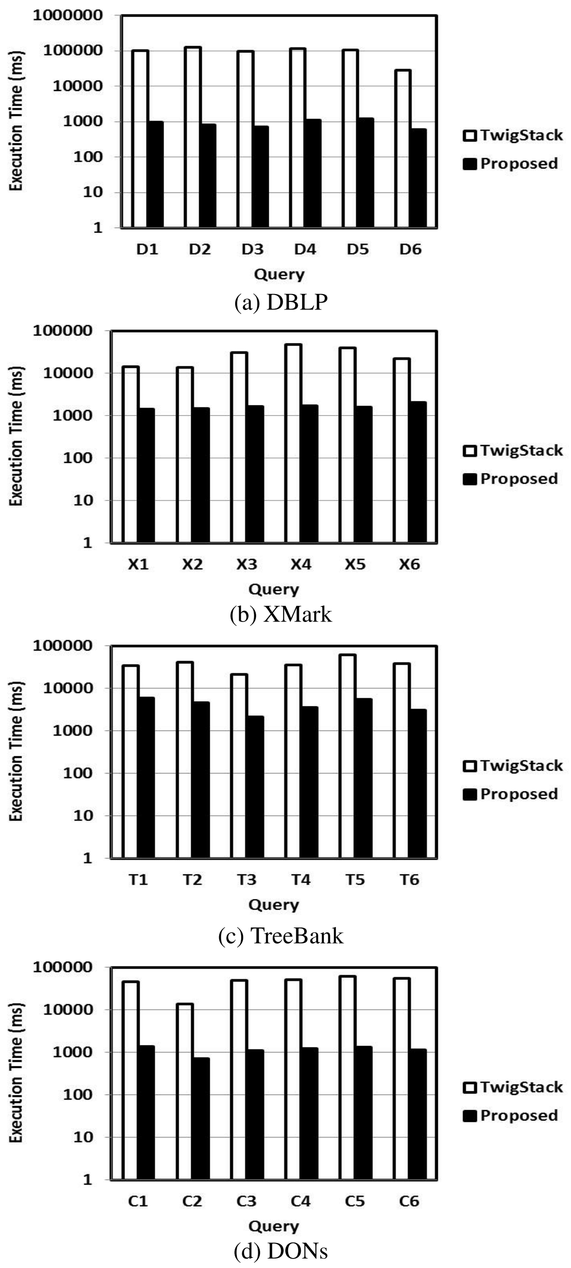
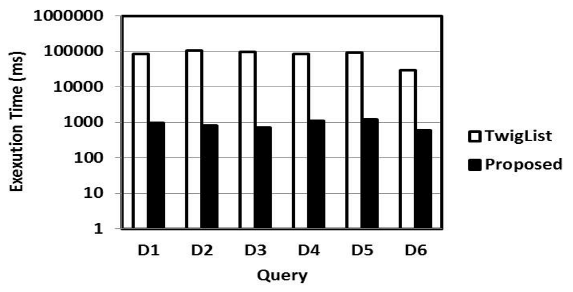
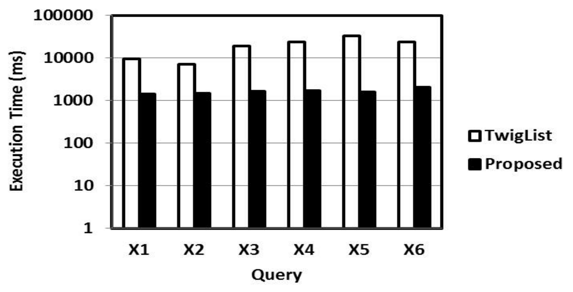


FIGURE 6. **Execution times of TwigStack and the proposed algorithm.**

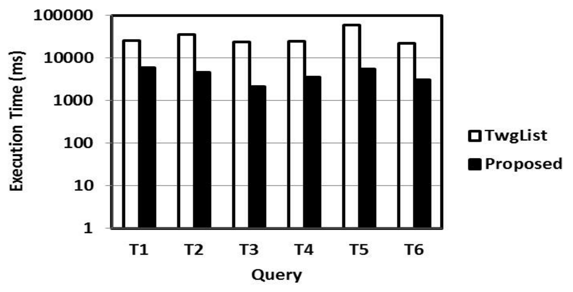
that correspond to the QTP leaf nodes. During query processing, TJFast performs expensive ancestors extraction, string matching, and path merging operations. It also generates large intermediate results and processes many data nodes which don't appear in the output of a given QTP. For shallow queries, the cost of ancestors extraction, string matching, and path merging might cancel the benefit of reading fewer node streams. MatchQTP outperforms TJFast because it reads only the data-stream of the target node, generates no intermediate results, and doesn't perform ancestors extraction, string matching, and path merging. Figures 8 show that MatchQTP significantly outperforms TJFast. The figure shows that, on the average, MatchQTP is 20 times faster than TJFast.



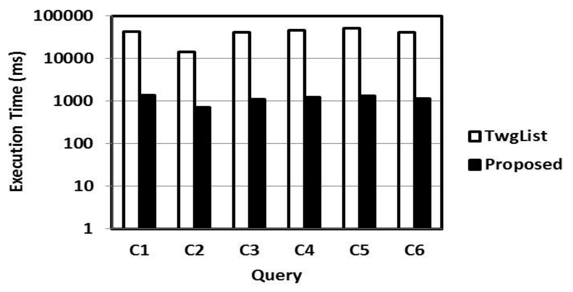
(a) DBLP



(b) XMark

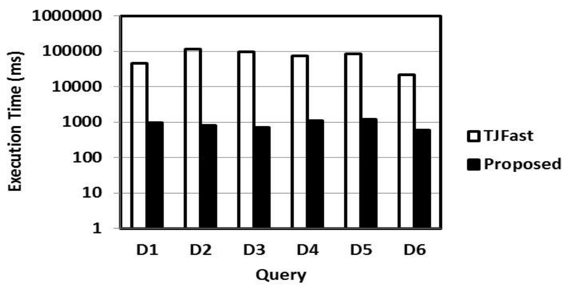


(c) TreeBank

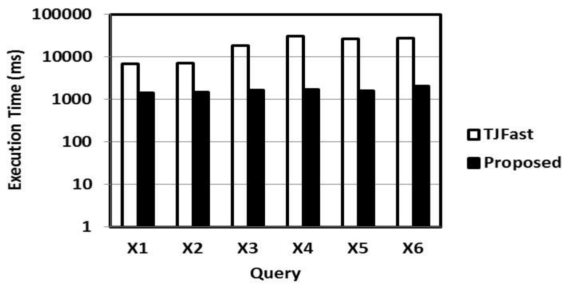


(d) DONs

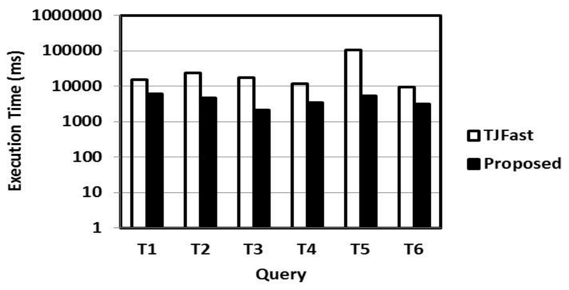
FIGURE 7. Execution times of TwigList and the proposed algorithm.



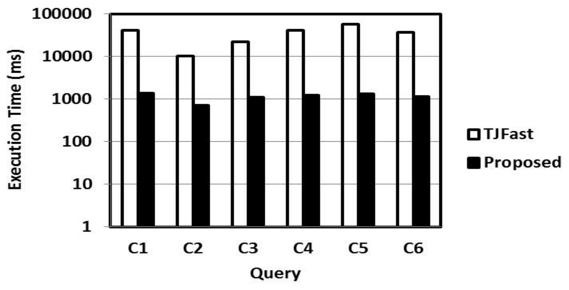
(a) DBLP



(b) XMark



(c) TreeBank

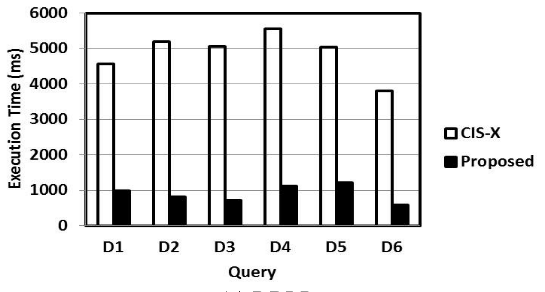


(d) DONs

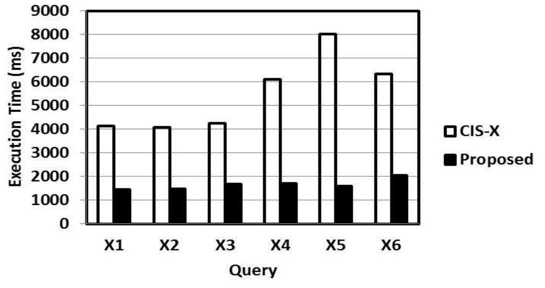
FIGURE 8. Execution times of TJFast and the proposed algorithm.

MatchQTP vs. CIS-X: CIS-X uses path-based streams. It processes many data-streams simultaneously and performance path merging. It also accesses many data-nodes that are not part of the final solution. MatchQTP processes one data-stream at time and thus no path merging is needed. It also doesn't access data-nodes that are not part of the final solution. Figure 9 shows the processing times of MatchQTP and CIS-X. On the average MatchQTP is 2.6 times faster than CIS-X.

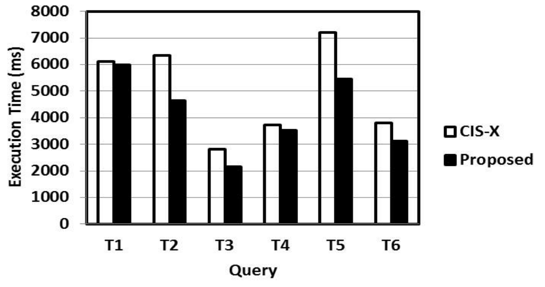
Cost of QTPs with value predicates: A number of experiments were conducted to study the performance of MatchQTP when evaluating QTPs with value predicates. The same QTPs shown in Table 6 were used for the experiment but one or two of the Qnodes were converted to value predicates. Let Q and $\hat{Q}$ be two structurally identical QTPs in which $\hat{Q}$ has value predicates whereas Q has none. In almost all the experiments, the cost of each Q was greater or comparable to that of $\hat{Q}$. The cost depended on the selectivity of $\hat{Q}$. If the selectivity is high, then the cost of $\hat{Q}$ was significantly lower than that of Q; and if the selectivity is very low, then the cost of Q and $\hat{Q}$ was nearly the same.



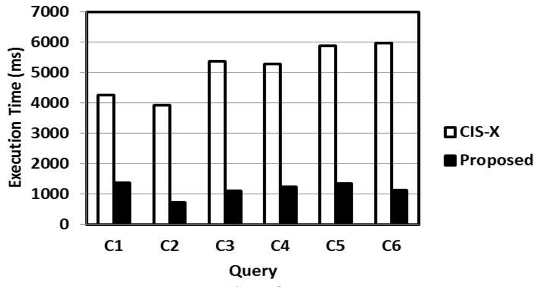
(a) DBLP



(b) XMark



(c) TreeBank



(d) DONs

FIGURE 9. Execution times of CIS-X and the proposed algorithm.

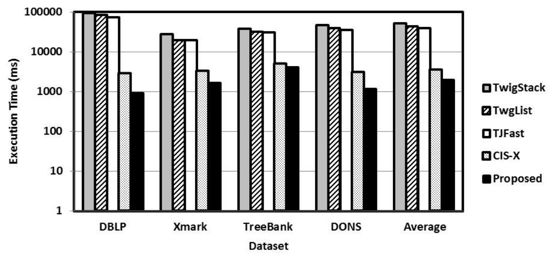
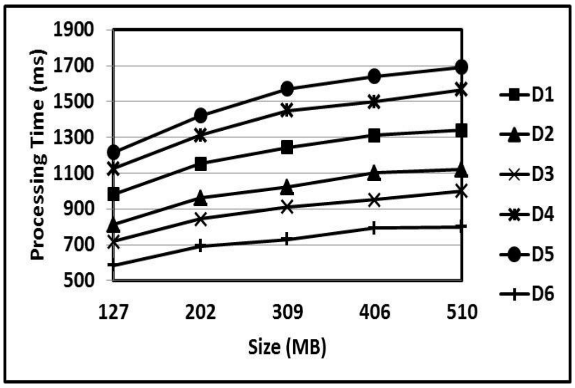
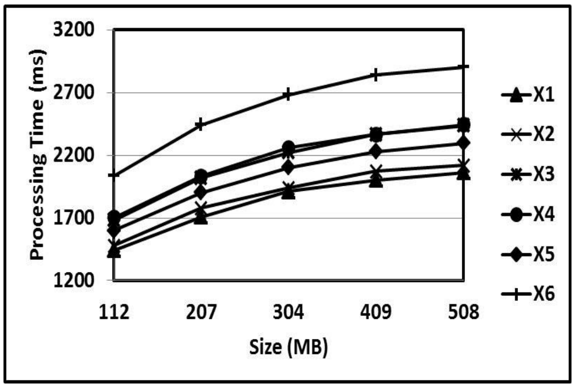


FIGURE 10. Average Processing time of MatchQTP.



(a) DBLP



(b) XMark

FIGURE 11. Scalability of MatchQTP.

Dataset structure effect on performance: The structure of a dataset affects the cost of processing QTPs using MatchQTP. During query processing, MatchQTP performs QTP matches only with DG and Ttrees. A dataset with many Ttrees results in more matching operations than one with smaller number of Ttrees. Each Ttree corresponds to one Dtree-stream; so a QTP which matches many Ttrees causes many random disk reads which adds to the I/O cost of the QTP. Another factor that adds to the cost of a query is the size of a DataGuide. On the average, finding QTP matches in a small DataGuide is less costly than in a bigger DataGuide. Datasets with small number of distinct nodes, few identical siblings, non-recursive paths, and many repeated subtree structures result in small DataGuide and few number of Ttrees. Among the 4 experimental datasets, DBLP has the smallest DataGuide and the smallest number of Ttrees. That is why the query cost of MatchQTP on a DBLP dataset is very low. The number of Ttrees in the DONS dataset is nearly the same as that of DBLP; but it has a bigger DataGuide and has many identical siblings. The average number of nodes in a DBLP Ttree is 9 whereas in DONS is 61. That is why query processing on a DONS dataset is more costly than on a DBLP dataset, 10. The number of Ttrees and the size of the DataGuide of the XMark dataset are higher than those of DONS. TreeBank has the highest number of Ttrees and the biggest DataGuide than the other 3 datasets. Figure 10 shows the average query cost on each dataset. The results show that DBLP has the lowest and TreeBank has the highest average query cost. The figure also shows that on the average

MatchQTP is 26, 22, 20, and 2.6 times faster than TwigStack, TwigList, TJFast, and CIS-X respectively.

Scalability of MatchQTP: To show the scalability of MatchQTP, a number of experiments were conducted on the DBLP and XMark datasets. The size of DBLP datasets ranged between 127 and 510 Megabytes and the size of XMark datasets ranged between 112 and 508 Megabytes. The queries listed in table 6 were used for the experiments. The results are shown in Figure 11. The sub-linear growth in processing time shows that MatchQTP is scalable. MatchQTP uses a DataGuide and Ttrees to find QTP matches. If the size of the DataGuide and the number of Ttrees remain the same, then the QTP processing time doesn't change much. The number of Ttrees and the size of a DataGuide depends on the structure of the XML-tree and not in the size of the dataset. If an XML document grows in size but its structure remains the same, then the size of its DataGuide and the number of its Ttrees will not change.

VIII. CONCLUSION

Internet of Things is an environment where smart services are enabled by smart devices and appliances. The value of these smart applications comes from analyzing the data generated by these smart devices and appliances. Handling this data efficiently is crucial to the success of IoT-enabled smart applications. Lack of efficient data handling renders the usability of IoT smart applications and hence IoT smart applications become unreliable. XML is frequently used by IoT devices to exchange data; thus, it is important to efficiently process XML data.

This paper proposes a new XML node labeling scheme called
RLP-Scheme, a new storage structure called RLP-Index, and a new twig pattern matching algorithm called MatchQTP. RLP-Index uses RLP-Scheme to partition data nodes into many streams. It stores a subset of the data-nodes. The rest of the data-nodes are stored implicitly. This makes it the first of its kind. This also minimizes storage space and QTP processing time.

MatchQTP, unlike the existing algorithms, processes one data-stream at a time, generates no intermediate results, and doesn't access data nodes which don't appear in the final solution of a given QTP. It doesn't perform any path merging or path extraction or duplicate removal operations. MatchQTP was compared with TwigStack, TwigList, TJFast, and CIS-X and it outperformed them consistently and significantly. MatchQTP is also scalable. Its QTP processing time grows sub-linearly with the size of the corresponding XML document.

As for future work, the proposed storage structure can be converted into a key-value store in which TID is the key and the DID is the value. Also MatchQTP can be converted to MapReduce algorithm with minor modification. We also envision that extending MatchQTP to handle QTPs with all logical operators will improve the efficiency process of XML data.

REFERENCES

[1] C. Gonzalez, S. M. Charfadine, O. Flauzac, and F. Nolot, "SDN-based security framework for the IoT in distributed grid," in *Proc. Int. Multidisciplinary Conf. Comput. Energy Sci. (SpliTech)*, Split, Croatia, Jul. 2016, p. 1–5.

[2] R. Zgheib, E. Conchon, and R. Bastide, "Semantic middleware architectures for IoT healthcare applications," in *Enhanced Living Environments*. Cham, Switzerland: Springer, 2019, pp. 263–294.

[3] F. Azzedin and M. Ghaleb, "Internet-of-Things and information fusion: Trust perspective survey," *Sensors*, vol. 19, no. 8, p. 1929, 2019.

[4] E. T. Heidt, "2017 planning guide for the Internet of Things," Gartner, Stamford, CT, USA, Tech. Rep. G00313353, Oct. 2016.

[5] *Db-Engines Ranking*. Accessed: Jun. 15, 2017. [Online]. Available: http://db-engines.com/en/ranking

[6] F. Merciol, A. Sauray, and S. Lefevre, "Interoperability of multiscale visual representations for satellite image big data," in *Proc. Conf. Big Data Space (BiDS)*. Santa Cruz de Tenerife, Spain, 2016, pp. 172–175.

[7] H. Cai, B. Xu, L. Jiang, and A. V. Vasilakos, "IoT-based big data storage systems in cloud computing: Perspectives and challenges," *IEEE Internet Things J.*, vol. 4, no. 1, pp. 75–87, Feb. 2017.

[8] W3C. *XPath*. Accessed: Jan. 1, 2019. [Online]. Available: https://www.w3.org/TR/xpath-31/

[9] W3C. *XQuery*. Accessed: Jan. 1, 2019. [Online]. Available: https://www.w3.org/TR/xquery-31/

[10] S.-C. Haw and C.-S. Lee, "Data storage practices and query processing in XML databases: A survey," *Knowl.-Based Syst.*, vol. 24, no. 8, pp. 1317–1340, Dec. 2011, doi: 10.1016/j.knosys.2011.06.006.

[11] M. Hachicha and J. Darmont, "A survey of XML tree patterns," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 1, pp. 29–46, Jan. 2013.

[12] G. Gou and R. Chirkova, "Efficiently querying large XML data repositories: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 10, pp. 1381–1403, Oct. 2007.

[13] H. Su-Cheng and L. Chien-Sing, "Node labeling schemes in XML query optimization: A survey and trends," *IETE Tech. Rev.*, vol. 26, no. 2, p. 88, 2009.

[14] T. Härder, M. Haustein, C. Mathis, and M. Wagner, "Node labeling schemes for dynamic XML documents reconsidered," *Data Knowl. Eng.*, vol. 60, no. 1, pp. 126–149, Jan. 2007.

[15] J. Lu, X. Meng, and T. W. Ling, "Indexing and querying XML using extended dewey labeling scheme," *Data Knowl. Eng.*, vol. 70, no. 1, pp. 35–59, Jan. 2011.

[16] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: Optimal xml pattern matching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: Association for Computing Machinery, 2002, p. 310–321, doi: 10.1145/564691.564727.

[17] J. Lu, T. Chen, and T. W. Ling, "TJFast: Effective processing of XML twig pattern matching," in *Proc. Special interest tracks posters 14th Int. Conf. World Wide Web WWW*, New York, NY, USA: ACM, 2005, pp. 1118–1119.

[18] S. Lee, B.-G. Ryu, and K.-L. Wu, "Examining the impact of data-access cost on XML twig pattern matching," *Inf. Sci.*, vol. 203, pp. 24–43, Oct. 2012.

[19] T. Chen, J. Lu, and T. W. Ling, "On boosting holism in XML twig pattern matching using structural indexing techniques," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, New York, NY, USA: ACM, 2005, pp. 455–466, doi: 10.1145/1066157.1066209.

[20] S. K. Izadi, M. S. Haghjoo, and T. Härder, "S3: Processing tree-pattern XML queries with all logical operators," *Data Knowl. Eng.*, vol. 72, pp. 31–62, Feb. 2012.

[21] W.-C. Hsu and I.-E. Liao, "CIS-X: A compacted indexing scheme for efficient query evaluation of XML documents," *Inf. Sci.*, vol. 241, pp. 195–211, Aug. 2013.

[22] S. K. Izadi, T. Härder, and M. S. Haghjoo, "s3: Evaluation of tree-pattern XML queries supported by structural summaries," *Data Knowl. Eng.*, vol. 68, no. 1, pp. 126–145, Jan. 2009.

[23] R. Bača and M. Krátkỳ, "Tjdewey—On the efficient path labeling scheme holistic approach," in *Proc. Int. Conf. Database Syst. Adv. Appl.* Brisbane, QLD, Australia: Springer, 2009, pp. 6–20.

[24] R. Bača and M. Krátký, "On the efficiency of a prefix path holistic algorithm," in *Proc. 6th Int. XML Database Symp. Database XML Technol. (XSym)*, Lyon, France: Springer-Verlag, 2009, pp. 25–32.

[25] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, "Covering indexes for branching path queries," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, New York, NY, USA: ACM, 2002, pp. 133–144.

[26] X. Wu and G. Liu, "XML twig pattern matching using version tree," *Data Knowl. Eng.*, vol. 64, no. 3, pp. 580–599, Mar. 2008.

[27] P. F. Dietz, "Maintaining order in a linked list," in *Proc. 14th Annu. ACM Symp. Theory Comput. STOC*, San Francisco, CA, USA, 1982, pp. 122–127.

[28] Q. Li and B. Moon, "Indexing and querying xml data for regular path expressions," in *Proc. 27th Int. Conf. Very Large Data Bases (VLDB)*, Roma, Italy, 2001, pp. 361–370.

[29] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On supporting containment queries in relational database management systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Santa Barbara, CA, USA, 2001, pp. 425–436.

[30] E. Cohen, H. Kaplan, and T. Milo, "Labeling dynamic XML trees," *SIAM J. Comput.*, vol. 39, no. 5, pp. 2048–2074, Jan. 2010.

[31] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, "Storing and querying ordered XML using a relational database system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, Madison, Wisconsin, 2002, pp. 204–215.

[32] M. Duong and Y. Zhang, "Lsdx: A new labelling scheme for dynamically updating xml data," in *Proc. 16th Australas. Database Conf.*, Newcastle, Australia, vol. 39, 2005, pp. 185–193.

[33] F. Weigel, K. U. Schulz, and H. Meuss, "The bird numbering scheme for xml and tree databases–deciding and reconstructing tree relations using efficient arithmetic operations," in *Proc. Int. XML Database Symp.* Trondheim, Norway: Springer, 2005, pp. 49–67.

[34] Y. K. Lee, S.-J. Yoo, K. Yoon, and P. B. Berra, "Index structures for structured documents," in *Proc. 1st ACM Int. Conf. Digit. Libraries DL*, Bethesda, Maryland, Mar. 1996, pp. 91–99.

[35] R. Al-Shaikh, G. Hashim, A. BinHuraib, and S. Mohammed, "A modulo-based labeling scheme for dynamically ordered XML trees," in *Proc. 5th Int. Conf. Digit. Inf. Manage. (ICDIM)*, Thunder Bay, ON, Canada, Jul. 2010, pp. 213–221.

[36] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugasundaram, and D. Srivastava, "Index structures for matching xml twigs using relational query processors," in *Proc. 21st Int. Conf. Data Eng. Workshops (ICDEW)*, Apr. 2005, p. 1273.

[37] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura, "XRel: A path-based approach to storage and retrieval of XML documents using relational databases," *ACM Trans. Internet Technol.*, vol. 1, no. 1, pp. 110–141, Aug. 2001, doi: 10.1145/383034.383038.

[38] N. Zhang, V. Kacholia, and M. T. Ozsu, "A succinct physical storage scheme for efficient evaluation of path queries in XML," in *Proc. 20th Int. Conf. Data Eng.*, Boston, MA, USA, Apr. 2004, pp. 54–65.

[39] P. Rao and B. Moon, "PRIX: Indexing and querying XML using prufer sequences," in *Proc. 20th Int. Conf. Data Eng.*, Boston, MA, USA, Apr. 2004, pp. 288–299.

[40] H. Wang, S. Park, W. Fan, and P. S. Yu, "ViST: A dynamic index method for querying XML data by tree structures," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, San Diego, CA, USA, 2003, pp. 110–121.

[41] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu, "Structural joins: A primitive for efficient XML query pattern matching," in *Proc. 18th Int. Conf. Data Eng.* San Jose, CA, USA, Mar. 2002, pp. 141–152.

[42] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo, "Efficient structural joins on indexed xml documents," in *Proc. 28th Int. Conf. Very Large Databases VLDB*. Hong Kong: Elsevier, Jan. 2002, pp. 263–274.

[43] H. Jiang, H. Lu, W. Wang, and B. C. Ooi, "XR-tree: Indexing XML data for efficient structural joins," in *Proc. 19th Int. Conf. Data Eng.*, Bangalore, India, Mar. 2003, pp. 253–264.

[44] J. Lu and T. W. Ling, "Labeling and querying dynamic xml trees," in *Proc. Asia–Pacific Web Conf.* Hangzhou, China: Springer, 2004, pp. 180–189.

[45] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen, "From region encoding to extended dewey: On efficient processing of xml twig pattern matching," in *Proc. 31st Int. Conf. Very large Databases (VLDB)*. Trondheim, Norway, 2005, pp. 193–204.

[46] N. Grimsmo, T. A. Bjørklund, and M. L. Hetland, "Fast optimal twig joins," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 894–905, Sep. 2010.

[47] R. Bača, M. Krátký, and V. Snášel, "On the efficient search of an xml twig query in large dataguide trees," in *Proc. Int. Symp. Database Eng. Appl. (IDEAS)*, New York, NY, USA, 2008, pp. 149–158, doi: 10.1145/1451940.1451962.

[48] F. Weigel, H. Meuss, F. Bry, and K. U. Schulz, "Content-aware dataguides: Interleaving ir and db indexing techniques for efficient retrieval of textual XML data," in *Proc. Eur. Conf. Inf. Retr.* Sunderland, U.K.: Springer, 2004, pp. 378–393.

[49] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in Internet of Things: Taxonomies and open challenges," *Mobile Netw. Appl.*, vol. 24, no. 3, pp. 796–809, Jun. 2019.

[50] Y. Zhang, W. Han, W. Wang, and C. Lei, "Optimizing the storage of massive electronic pedigrees in HDFS," in *Proc. 3rd IEEE Int. Conf. Internet Things*, Wuxi, China, Oct. 2012, pp. 68–75.

[51] M. Li, Z. Zhu, and G. Chen, "A scalable and high-efficiency discovery service using a new storage," in *Proc. IEEE 37th Annu. Comput. Softw. Appl. Conf.*, Kyoto, Japan, Jul. 2013, pp. 754–759.

[52] A. Markus, G. Kecskemeti, and A. Kertesz, "Flexible representation of IoT sensors for cloud simulators," in *Proc. 25th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*. St. Petersburg, Russia, Mar. 2017, pp. 199–203.

[53] Y. Xu and T. Kishi, "An ontology-based IoT communication data reduction method," in *Proc. 9th IEEE Annu. Ubiquitous Comput., Electron. Mobile Commun. Conf. (UEMCON)*, New York City, NY, USA, Nov. 2018, pp. 321–325.

[54] H.-W. Kim, J. H. Park, and Y.-S. Jeong, "Efficient resource management scheme for storage processing in cloud infrastructure with Internet of Things," *Wireless Pers. Commun.*, vol. 91, no. 4, pp. 1635–1651, Dec. 2016.

[55] K. R. Malik, Y. Sam, M. Hussain, and A. Abuarqoub, "A methodology for real-time data sustainability in smart city: Towards inferencing and analytics for big-data," *Sustain. Cities Soc.*, vol. 39, pp. 548–556, May 2018.

[56] P. K. Gupta, B. T. Maharaj, and R. Malekian, "A novel and secure IoT based cloud centric architecture to perform predictive analysis of users activities in sustainable health centres," *Multimedia Tools Appl.*, vol. 76, no. 18, pp. 18489–18512, Sep. 2017.

[57] S. Ahmad, L. Hang, and D. Kim, "Design and implementation of cloud-centric configuration repository for DIY IoT applications," *Sensors*, vol. 18, no. 2, p. 474, 2018.

[58] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," in *Proc. 23rd Int. Conf. Very Large Data Bases (VLDB)*, San Francisco, CA, USA: Morgan Kaufmann, 1997, pp. 436–445. [Online]. Available: http://dl.acm.org/citation.cfm?id=645923.671008

[59] L. Qin, J. X. Yu, and B. Ding, "Twiglist: Make twig pattern matching fast," in *Proc. 12th Int. Conf. Database Syst. Adv. Appl.* Bangkok, Thailand: Springer, 2007, pp. 850–862.

[60] *DBLP: Digital Bibliography & Library Project*. Accessed: Mar. 1, 2018. [Online]. Available: http://dblp.uni-trier.de/xml/

[61] *University of Washington Database Group: The XML Data Repository*. Accessed: Mar. 1, 2018. [Online]. Available: http://www.cs.washington.edu/research/xmldatasets/

[62] *XMark—An XML Benchmark Project*. Accessed: Mar. 1, 2018. [Online]. Available: http://www.xml-benchmark.org/

[63] F. Azzedin, J. Yazdani, S. Adam, and M. Ghaleb, "A generic model for disease outbreak notification systems," *Int. J. Comput. Sci. Inf. Technol.*, vol. 6, no. 4, pp. 137–154, Aug. 2014.

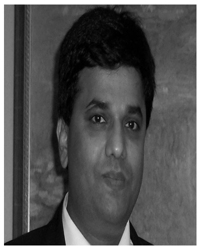
[64] F. Azzedin, S. Mohammed, J. Yazdani, and M. Ghaleb, "Designing a disease outbreak notification system in Saudi Arabia," in *Proc. 2nd Int. Conf. Adv. Comput. Sci. Inf. Technol. (ACSIT)*, Zurich, Switzerland, 2014, pp. 1–14.

[65] F. Azzedin, S. Mohammed, J. Yazdani, T. A. Ghaleb, and M. Ghaleb, "A cloud-based prototype implementation of a disease outbreak notification system," *Int. J. Comput. Sci., Eng. Appl.*, vol. 5, no. 2, pp. 15–31, Apr. 2015.

FARAG AZZEDIN received the B.Sc. degree in computer science from the University of Victoria, Canada, and the M.Sc. and Ph.D. degrees in computer science from the Computer Science Department, University of Manitoba, Canada. He is currently an Associate Professor with the Department of Information and Computer Science, King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia.

SALAHADIN MOHAMMED received the B.S. and M.S. degrees in computer science from the King Fahd University of Petroleum and Minerals (KFUPM) and the Ph.D. degree in computer science from the Computer Science Department, Monash University, Melbourne, Australia. He is currently an Assistant Professor with the Department of Information and Computer Science, KFUPM.

MUSTAFA GHALEB was born in Taiz, Yemen, in 1983. He received the B.S. degree in computer from King Khalid University (KKU), Abha, Saudi Arabia, in 2007, and the M.S. degree in information and computer sciences from the King Fahd University of Petroleum and Minerals (KFUPM). Dhahran, Saudi Arabia, in 2015, where he is currently pursuing the Ph.D. degree in information and computer sciences.



JAWEED YAZDANI received the B.S. degree in computer science and the M.S. degree from the King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia. He is currently a faculty member with the Department of Information and Computer Science, KFUPM, and a Manager of the Administrative Information Systems (ADIS), KFUPM.

ADEL AHMED received the B.S. degree in mathematics from King Saud University, in 1995, the M.S. degree in computer science from the King Fahd University of Petroleum and Minerals (KFUPM), Saudi Arabia, in 2001, and the Ph.D. degree in computer science from The University of Sydney, Australia, in 2008. He is currently the Dean of the College of Computer Science and Engineering, KFUPM.