Data & Knowledge Engineering xxx (2008) xxx–xxx

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

S^3: Evaluation of tree-pattern XML queries supported by structural summaries ☆

Sayyed Kamyar Izadi [a], Theo Härder [b,*], Mostafa S. Haghjoo [a]

[a] Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
[b] Department of Computer Science, University of Kaiserslautern, D-67663 Kaiserslautern, Germany

A R T I C L E I N F O

Article history:
Received 9 June 2008
Received in revised form 8 August 2008
Accepted 2 September 2008
Available online xxxx

Keywords:
XML query processing
Twig queries
Query guide
Tree-pattern matching
Structural summary
DeweyIDs

A B S T R A C T

XML queries are frequently based on path expressions where their elements are connected to each other in a tree-pattern structure, called query tree pattern (QTP). Therefore, a key operation in XML query processing is finding those elements which match the given QTP. In this paper, we propose a novel method, called S^3, which can selectively process the document's nodes. In S^3, unlike all previous methods, path expressions are not directly executed on the XML document, but first they are evaluated against a guidance structure, called *QueryGuide*. Enriched by information extracted from the *QueryGuide*, a query execution plan, called SMP, is generated to provide focused pattern matching and avoid document access as far as possible. Moreover, our experimental results confirm that S^3 and its optimized version OS^3 substantially outperform previous QTP processing methods w.r.t. response time, I/O overhead, and memory consumption – critical parameters in any real multi-user environment.

© 2008 Published by Elsevier B.V.

1. Introduction

XML usage is increasing dramatically. There are many applications such as science, biology, business, and, particularly, web information systems using XML as their data representation format. This growing trend towards XML confirms the need of XML database management systems (XDBMSs). Query processing is an essential functionality of any DBMS; this is especially challenging for XDBMSs, because XML documents combine tree structure with content. Both XPath and XQuery, the two most popular query languages in the XML domain, are based on path expressions. A so-called query tree pattern (QTP) specifies a pattern of selection predicates addressing multiple elements in a path expression related by a tree structure. As a focal point of our discussion, these patterns include the most important query axes *parent–child* and *ancestor–descendant* (*P–C* or / and *A–D* or //, for short). To process XML queries, all fragments matching a QTP in the XML document have to be found, which is an expensive task, especially when huge XML documents are involved.

Consider the following query Q_1 addressing a given XML document: Q_1: *//A[.//B]/C//D*. Q_1's QTP which is shown in Fig. 1 has two branches, *A//B* and *A/C//D*. The elements related to Q_1 could be easily found via traditional indexes like B^*-trees, but such an access support to select elements is not enough, because the located elements must satisfy the path conditions,

☆ This work has been partially done while the first author was visiting at the University of Kaiserslautern. It has been supported by the Iran University of Science and Technology, Tehran, Iran, and the Rheinland-Pfalz cluster of excellence "Dependable adaptive systems and mathematical modeling", Germany (see www.dasmod.de).
* Corresponding author. Tel.: +49 631 205 4030; fax: +49 631 205 3299.
E-mail address: haerder@informatik.uni-kl.de (T. Härder).
URL: http://wwwlgis.informatik.uni-kl.de/cms/index.php?id=30 (T. Härder).

0169-023X/$ - see front matter © 2008 Published by Elsevier B.V.
doi:10.1016/j.datak.2008.09.001

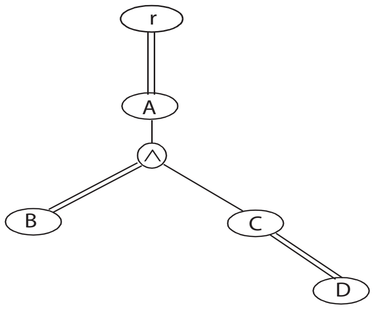

Fig. 1. Related QTP of Q_1.

too. Therefore, the most important issue during query evaluation is to check that all extracted elements meet the given path expression, before they are composed to the desired query result. This process may become even more complicated and costly when more than two branches occur in a path expression. The question, how to optimally locate the fragments matching a given QTP (such as that of Q_1) in an XML document, attracted many researchers, e.g., see [1,4,10,15,22].

1.1. Our contribution

The key observation to optimize XML query evaluation is to avoid document access as far as possible. For this reason, we introduce a so-called *QueryGuide* which is an abstraction of the considered XML document. It is a kind of structural summary and describes the document structure by its path classes which, given the node number (in the form of a Dewey label) of any document node, enables the reconstruction of the specific path instance the considered node belongs to. In this way, the *QueryGuide* serves to especially present the *P–C* and *A–D* relationships of the elements in an XML document and to support the matching process based on the QTP, that is, it *guides the query evaluation* to enable focused search.

Hence, the interplay of Dewey node labeling and *QueryGuide* use leads to a new quality of QTP processing which is captured by the main contributions of our paper:

- The concept SMP (set of match patterns) is introduced to enable focused node comparisons and to facilitate path checking.
- To create an SMP, we use a structure called *QueryGuide* which acts as a structural summary of the XML document.
- We present some optimization hints for S^3 leading to Optimized S^3 (OS^3).

The remaining parts of our paper are organized as follows: Section 2 provides background information needed for the rest of paper and gives an overview of the most important related approaches. The concepts of Dewey labels and *QueryGuide* which are the cornerstones for S^3 are discussed in Section 3. We introduce S^3 and its optimized version OS^3 in Section 4. In Section 5, we present the experimental results and conclude our work in Section 6.

2. Basic concepts and related approaches

In this section, we present a simplified logical representation for XML documents based on the standard XPath data model [2], before we sketch and compare five approaches competing with our novel method proposed.

2.1. Data and QTP model

So far, quite a number of differing approaches to QTP processing on XML documents were developed. To facilitate their description and classification, we need an appropriate terminology and some important definitions. Also a more formal definition of QTP is given.

Definition 1. An XML tree structure (XTS) X is a tree defined by a tuple (r, N_X, E, I, T, V):

- $r \in N_X$ as an auxiliary node is the root of the XML tree.
- N_X is a set of XTS nodes.
- $E \subset N_X \times N_X$ represents relations between nodes (branches of the tree).
- $I : N_X \rightarrow String$ is a function returning the unique label of the requested node.
- $T : N_X \rightarrow$ {"root", "element", "attribute", "text"} is a function which returns the type of a node.
- $V : N_X \rightarrow String$ is a function which returns the value of a node. "root" is the value assigned for the auxiliary root of the XML tree (i.e., $V(r)$ = "root").

Definition 2. A QTP[1] is a tree structure defined by the tuple $(r'', Q, O, E'', U, V'', C)$ over an XTS object X:

- $r'' \in Q$ is the root of the QTP.
- Q is a set of *query nodes* in the QTP defined as follows: $Q = \{x | \exists n \in N_X, T(n) = \text{“element”} \vee T(n) = \text{“attribute”}, V(n) = V''(x)\} \cup r''$.
- $O = \wedge, \vee, \neg, \oplus$ is a set of logical operator nodes in a QTP. $(\wedge, \vee, \oplus)$ represent the binary AND, OR, and XOR logical operators, respectively. $(\neg)$ is the unary NOT operator.
- $E'' \subset (Q \cup O) \times (Q \cup O)$ represents branches of QTP. All leaves of the QTP are query nodes.
- $U : Q \times \{\text{“A–D”}, \text{“P–C”}\}$ indicates a kind of relationship between a query node q and its nearest query node among ancestors of q. “P–C” shows a parent–child $(/)$ relationship, while “A–D” represents an ancestor–descendant $(//)$ relationship between nodes of the QTP, which has to be satisfied during the matching process over the associated XTS object X.
- $V'' : Q \to String$ returns the value of a node. “root” is the value assigned to the root of the QTP ($V''(r'')$ = “root”).
- $C : Q \times N_X \to \{true, false\}$ is a Boolean function deciding whether or not a node $n \in N_X$ satisfies the content constraints associated with a query node q.

Definition 3. The potential target nodes (PTN) of a query node q in QTP$(r'', Q, O, E'', U, V'', C)$ defined over the XTS object $X(r, N_X, E, I, T, V)$ are contained in an ordered list of X nodes (PTN, <):

(1) $PTN(q) = \{n | n \in N_X, V(n) = V''(q) \wedge C(q, n)\}$.
(2) $\forall n_1, n_2 \in N_X : n_1 < n_2$ iff n_1 is visited earlier than n_2 in a pre-order traversal through the X.

Example 1. Fig. 2a is a simple XML document which is represented as an XML tree structure[2] (X_1) in Fig. 2b. Now consider the following XPath query:

$$Q_2 : //A//M//B[\text{contains}(\cdot, \text{“text5”})].$$

It is straightforward to see from Fig. 2b that the PTN of query node A is the set $\{a_1, a_2, a_3, a_4\}$ and the PTN of query node M is the set $\{m_1, m_2, m_3, m_4\}$, but m_3 is the only potential target node of query node B, because b_1 and b_3 do not satisfy the constraint associated with node B in Q_2. Obviously, (a_3, m_3, b_2) is the only match for Q_2. Hence, in this case, most of the potential target nodes of Q_2 are useless.

This observation motivated many researchers to develop evaluation methods for path expressions accessing nodes as few as possible.

2.2. Important methods for QTP evaluation

In this section, we take a quick look over some well-known QTP processing methods which have been developed in recent years. *Structural Join* is one of the first methods proposed to process XML path expressions [1]. By this method, path expressions are decomposed into several binary P–C or A–D relationships where each binary relationship is separately executed and its intermediate result is stored for further processing. The final result is formed by combining these intermediate results. For example, in order to process query Q_1 over X_1, Q_1 is decomposed into its three basic relationships $(A//B, A/C, C//D)$. Their independent evaluation delivers three intermediate results, as depicted in Fig. 3a. While the result for the first leg of Q_1 is already complete, the result for the second leg $A/C//D$ has to be derived by combining two intermediate lists. The final result is eventually gained by “intersecting” the result lists of both legs according to the QTP expression. Assume all nodes referenced in the evaluation have to be located in the physical XML document representation and fetched from external storage. Obviously, Structural Joins cause substantial overhead even for simple queries like Q_1. We can easily infer that more complicated queries, even on middle-sized documents, would perform much worse and derive huge volumes of intermediate results which are not needed for the final result. For example, isolated execution of the partial query expression $C//D$ produces many pairs of c and d elements, e.g., (c_1, d_1), while it is not guaranteed that for each pair of c and d elements a related pair of a and c elements could be found in the intermediate result of the partial query expression A/C. Some methods like those proposed in [8,16] attempt to improve the efficiency of the *Structural Join* using index structures.

Another often referenced QTP processing method is *TwigStack* [4]. It is a two-phase algorithm which does not decompose a query into its basic relationships. Instead, partial solutions for each leg (root-to-leaf path) in the QTP are found in the first phase of the algorithm. For example, Fig. 3b shows two intermediate sets which are produced if each leg of Q_1 is separately executed over X_1. It is worth noting that, in this phase, *TwigStack* is not able to check P–C relationships. As a consequence, some false positive results may occur in an intermediate result set like (a_2, c_3, d_2) for $A/C//D$ (a_2 is not the parent of c_3). Such false positives have to be removed prior to the subsequent phase of the algorithm to limit useless processing. *TwigStack* produces final matches in its second phase by merging single path results together using a merge-join algorithm. In order to

[1] In this paper, we focus on QTPs which only contain logical AND connectors and do not consider predicates, because we want to compare our method with competing ones based on the number of input nodes.

[2] Upper-case letters identify given query nodes and *QueryGuides* (explained in Section 3.2), whereas lower-case letters are used for their elements in the XML document. To preserve readability, we use q (lower-case letter) for the generic query node.

a

```
<x>
  <c><d>text1</d></c>
  <a><c>text2</c></a>
  <a>
    <m><b>text3</b></m>
    <m><c><d>text4</d></c></m>
  </a>
  <a>
    <m><b>text5</b></m>
    <c><m><d>text6</d></m></c>
  </a>
  <c><a><b>text7</b></a></c>
  <a><m><b>text8</b></m></a>
</x>
```

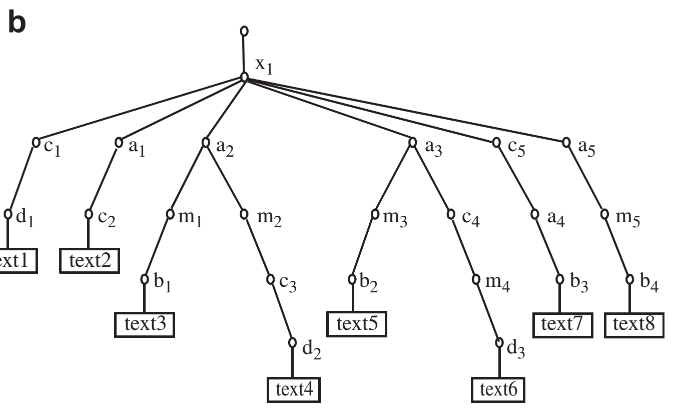

Fig. 2. (a) A simple XML document; (b) XTS representation of the document (X_1).

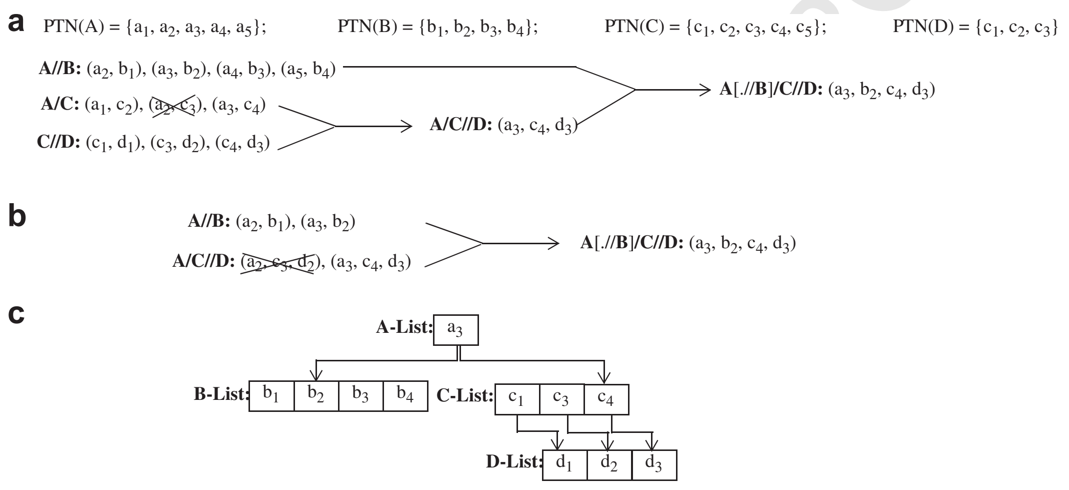
a
PTN(A) = {a1, a2, a3, a4, a5};
PTN(B) = {b1, b2, b3, b4};
PTN(C) = {c1, c2, c3, c4, c5};
PTN(D) = {c1, c2, c3}
A//B: (a2, b1), (a3, b2), (a4, b3), (a5, b4)
A/C: (a1, c2), ~~(a2, c3)~~, (a3, c4)
C//D: (c1, d1), (c3, d2), (c4, d3)
A/C//D: (a3, c4, d3)
A[.//B]/C//D: (a3, b2, c4, d3)
b
A//B: (a2, b1), (a3, b2)
A/C//D: ~~(a2, c3, d2)~~, (a3, c4, d3)
A[.//B]/C//D: (a3, b2, c4, d3)
c
A-List: a3
B-List: b1 b2 b3 b4
C-List: c1 c3 c4
D-List: d1 d2 d3

Fig. 3. Processing of Q_1 over X_1 (a) *Structural Join*; (b) *TwigStack*, *TJFast*; (c) *TwigList*.

decrease the cost of the merging phase, *TwigStack* outputs only those single path results in the first phase which have a chance to be joined with other path results to form a complete match. Comparing Fig. 3a and b clearly indicates that the amount of intermediate results derived by *TwigStack* is usually much lower than that of *Structural Join*. To improve *TwigStack*, index use was proposed [6,7,15]. *TwigStackList* [18] tries to solve the problem of false positives in intermediate *TwigStack* results by a look-ahead approach, whereas *TwigOptimal* [10] tries to achieve better performance by jumping during the evaluation process over non-qualified elements in the indexes.

Inspired by *TwigStack*, the *TJFast* [19] algorithm varies the idea of processing an entire leg of the QTP at a time. Hence, also running in two phases, *TJFast* only accesses potential target nodes of QTP leaves thereby minimizing its I/O requirements in the first processing phase. To achieve this improvement, *TJFast* uses a refined version of the Dewey labeling method (see also Section 3.1), which encodes the complete ancestor path in the label of each node. Furthermore, *TJFast* uses a finite state transducer (FST) to compute the complete path of a document node from its label. Thus, *TJFast* can easily produce partial results of individual root-to-leaf paths of the query only by accessing the potential target nodes of QTP leaves. *TJFast* derives the complete paths of the nodes accessed by translating their labels and subsequently produces the possible solutions for each leg of the QTP. These intermediate results are then merged together to form the query result. This second phase of the algorithm coincides with that of *TwigStack*.

Twig^2*Stack* [5] and its refined version, *TwigList* [22], are two other QTP processing methods, which primarily aim at the elimination of the merge cost in the second phase of *TwigStack* or *TJFast*. By these methods, intermediate results found while accessing the referenced nodes of the document are kept in such a way that subsequent merging is avoided and final matches are ready to be output only by applying a simple enumeration function. But these methods suffer from a severe weakness: they have to load the entire document into main memory in the worst case. *TwigList* tries to solve this problem by offering an external version which maintains intermediate results on external storage instead of main memory. In fact,

Table 1
Comparison of XML path expression processing methods

Aspect	Structural Join	TwigStack	TJFast	Twig^{2}Stack	TwigList	Our method
# of PTNs	All	All	Leaves	All	All	Leaves
# of elements	~# of index accesses					Minimum
Intermediate results	Large	~# of results				Insignificant

the main difference between both methods is that *TwigList* has only changed the complicated stacks used in *Twig^{2}Stack* to simpler list structures. Thus, because of this simpler mapping, *TwigList* can be suitably represented on external storage. Fig. 3c sketches the state of the lists used in *TwigList* when Q_1 is executed over X_1 prior to enumerating the final result, which is easily produced by traversing these chained lists. However, it is clear from Fig. 3c that *TwigList* has to load all d and b elements into its *D-List* and *B-List*, while only d_3 and b_2 are useful in our evaluation example.

In a nutshell, to identify the optimal method for practical applications, the above mentioned algorithms will be compared with our own proposal where the following three parameters play the major role: number of PTN sets that need to be accessed, number of elements which have to be read, and amount of *intermediate results* produced. Table 1 qualitatively compares these three parameters.[3] Note, all methods except *TJFast* and our own method have to access all PTNs related to the QTP nodes. We will show that the number of elements which have to be read is minimal in our method, even when compared to *TJFast*. *Structural Join* produces the maximum amount of intermediate results, which are insignificant for our method. Also none of the above listed query processing methods exploits the full potential of path indexes or summaries guiding the query execution. Exploiting more expressive node labeling based on the Dewey labeling method and a so-called *QueryGuide*, we can avoid document access for the query evaluation to the extent possible.

3. Key ingredients for the evaluation of XML path expressions

The power of our method is founded upon two key concepts: DeweyIDs and *QueryGuide*. In this section, we introduce these two concepts.

3.1. Node labeling

An intensive comparison of labeling schemes and their empirical evaluation [12] led us to use a prefix-based scheme for the labeling of tree nodes derived from the concept of Dewey order encoding. Dewey labeling was first used in libraries to make items easier to find on the shelves [9]. Dewey labels in the XML database domain consist of a sequence of so-called divisions (separated by dots in the human readable format) and represent the path from the document's root to the labeled node and the local order w.r.t. the parent node; in addition, optional sparse numbering facilitates node insertions and deletions [12]. If node u is the nth child of node v in a given XTS object, then:

$$Dewey(u) = Dewey(v) + \text{'.'} + f(n).$$

The Dewey label of the document's actual root is always set to 1. $f(n)$ is used to assign order-preserving values to the child labels. Hence, as shown by this construction principle, the label of each node contains the labels of all its ancestors. Fig. 4 represents X_1 which is labeled by the Dewey order encoding scheme (text nodes are not shown). In this example, $f(n)$ is set to $2n - 1$ for simplicity.[4]

Refining this idea, several similar labeling schemes were proposed which differ in some aspects such as overflow technique for dynamically inserted nodes, attribute node labeling, or encoding mechanism. Examples of such schemes are DLNs [3] or OrdPaths [21] developed for the Microsoft SQL Server™. Although similar to them, our mechanism is characterized by some distinguishing features and a label is denoted DeweyID [12]; it refines the Dewey order mapping with a *dist* parameter used to increment division values to leave gaps in the numbering space between consecutive labels – a kind of adjustment to expected update frequencies – and introduces an overflow mechanism when gaps for new insertions are in short supply. Any prefix-based scheme is appropriate for our document storage and QTP processing method and embodies the key to efficiency for other internal XML processing tasks[5] [14].

Existing DeweyIDs are immutable, that is, they allow the assignment of new IDs without the need to reorganize the IDs of nodes present. Comparison of two DeweyIDs allows ordering of the respective nodes in document order. Furthermore, DeweyIDs easily provide the labels of all ancestors. For example, the ancestor IDs of node d_2 with DeweyID 1.5.3.1.1 are 1.5.3.1, 1.5.3, 1.5, and 1.

[3] Compared to all methods, the maximum intermediate results denoted as *large* are produced by the *Structural Join*. The term *insignificant* means that they are minimal w.r.t. the other methods and that the volume is marginal in most realistic cases.
[4] In fact, the labels used in Fig. 4 consist of two parts: the first part is named CID (to be explained in Section 3.2) and the second part is the DeweyID.
[5] The term SPLIDs (Stable Path Labeling IDentifiers) is used in [24] as a synonym for labels constructed according to any prefix-based scheme.

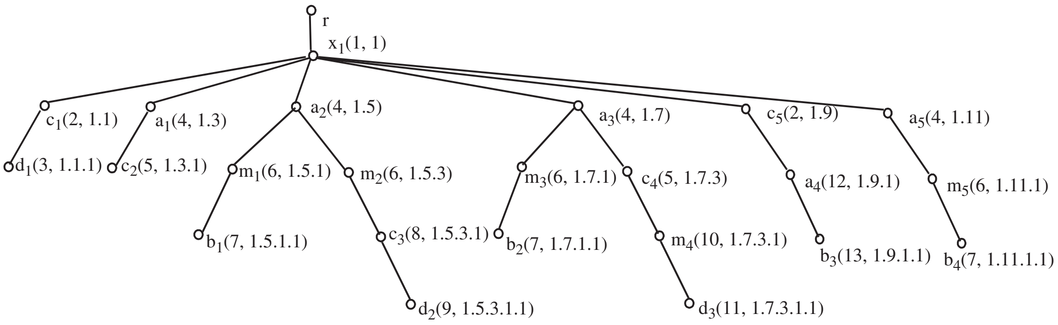

Fig. 4. X_1 labeled by Dewey order encoding.

The ability to derive all ancestor labels from a node's label is a valuable property, because it can provide the labels of potential target nodes for each query node in a given QTP by accessing only the labels of potential target nodes of the QTP leaves. This is the main idea, which is also used in *TJFast* to reduce I/O cost. We will show in the next section how we can minimize I/O cost by using DeweyIDs in conjunction with our document summarizer, called *QueryGuide*.

3.2. Summarization of XML documents

Querying huge amounts of data is an expensive task. Indexes like B^*-trees are solutions facilitating this process. Because of the mixture of structure and content in a tree-based data model – such as that for XML documents – it is obvious that traditional indexes cannot satisfy all needs of efficient query processing. Traditional methods can only index values of an XML document, but not paths to elements. Therefore, it is reasonable to additionally capture (or summarize) the structure of XML documents in a specific path index to facilitate evaluation of XML queries.

The idea of structure summarization is not a completely new idea. DataGuide [11] is a structural summary proposed for semi-structured documents. Its main purpose was to provide a structural overview to facilitate the formulation of meaningful queries and to store statistical document information to be used for query optimization. Furthermore, similar structures called path synopses were used to virtualize the structure part of documents [13]. In contrast, our goal of employing a structure summarizer is more than only accessing nodes faster or avoiding the explicit storage of the inner document nodes. Structural summaries or path indexes contain an abstraction of XML documents which can help us to have a more efficient query evaluation method for complicated XML queries where the QTPs may have more than one branch.

The *QueryGuide* is our data structure developed to summarize the XML document structure. During the evaluation of XML queries, a *QueryGuide* not only enables more focused access to document nodes which leads to I/O minimization, but also provides execution plans to process the referenced nodes in a more efficient way. Some simple definitions facilitate the introduction of the *QueryGuide* concept:

Definition 4. Considering an XTS object X and a node n, $n \in N_X$, a *traversal path* of node n is $P(n)$ consisting of all ancestors of node $n(a_1 \cdot a_2 \cdots a_j \cdot n)$ ordered by the P–C relationship. With regard to the traversal path of node n, $PS(n)$ is the relevant path string of node n represented as $/V(a_1)/V(a_2)/\cdots/V(a_j)/V(n)$.

Definition 5. Two traversal paths $P(n)$ and $P(m)$ are *path equivalent* ($P(n) \equiv P(m)$), if $PS(n) = PS(m)$. Also, nodes n and m are path equivalent if their traversal paths are path equivalent.

Example 2. Consider the two nodes c_2 and c_4 labeled by DeweyIDs 1.3.1 and 1.7.3 in Fig. 4. They are path equivalent as they have the same traversal path string "$/x/a/c$". Now consider node c_3 with DeweyID 1.5.3.1. This node is not path equivalent to c_2 and, thus, to c_4, because the path string of c_3 is not the same as the path string of c_2 ("$/x/a/c$" $\neq$ "$/x/a/m/c$").

Predefinition 1. A *Structural Summary* is an XTS object S which is defined over an XTS object X, represented by S_X.

Definition 6. In order to distinguish between traversal paths in a *Structural Summary* and its related XTS object, we refer to paths in a *Structural Summary* as *path classes* and paths in an XTS object as *path instances*. Also, nodes in the *Structural Summary* are referred to as *class nodes* which have a unique *CID (Class ID)* label and nodes in an XTS object are referred to as *instance nodes.*

Definition 7. Consider *Structural Summary* S_X and its XTS object X, the instance node set of a class node m, $m \in N_S$, is $INS(m) = \{n | n \in N_X, P(n) \equiv P(m)\}$.

Example 3. In Fig. 5a, the instance node set of node M with CID 6 (M_6) is the set $\{m_1, m_2, m_3, m_5\}$ and that of node M with CID 10 (M_{10}) is the set $\{m_4\}$.

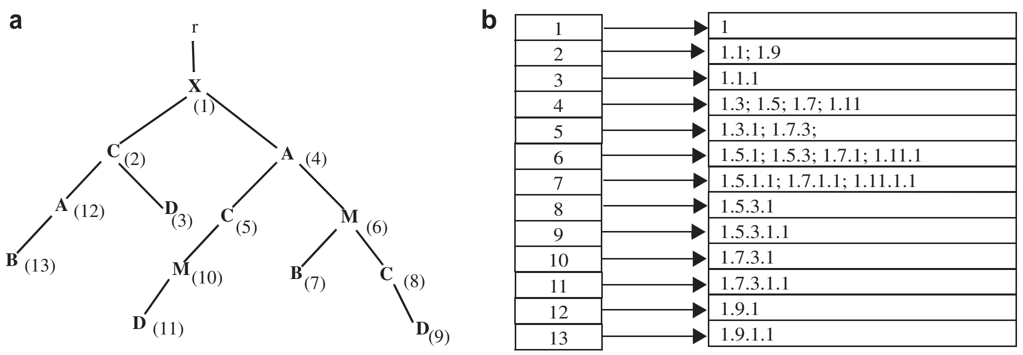

Fig. 5. *QueryGuide* for X_1 in Fig. 4: (a) *Structural Summary* (S_1); (b) reference section.

Definition 8. Considering an XML object X and its related *Structural Summary* S_X, the label of a given node n, $n \in N_X$, is the pair (c, d) where d is the DeweyID of n in X and c is the CID of node m, $m \in N_S$, such that $n \in INS(m)$.

Example 4. The CID of node m_4 in Fig. 4 is set to 10, because m_4 is path equivalent to node M_{10} in the *Structural Summary* of Fig. 5a ($PS(m_4) = PS(M_{10}) =$ "$/x/a/c/m$").

With respect to Definition 8, we are able to illustrate one of the greatest advantages of our labeling method: For a given node with label (c, d), the whole label and also the names of all of its ancestors could be extracted on the fly without accessing any other nodes. For example, consider node m_4 in Fig. 4 with label (10, 1.7.3.1). A quick look to the *Structural Summary* of Fig. 5a shows that the ancestors of m_4 are $x(1,1)$, $a(4,1.7)$, and $c(5,1.7.3)$.

Definition 9. The *Structural Summary* S_X is an XTS object defined over an XTS object X such that:

- $\forall n \in N_X$, $T(n) \neq$ "text" then $\exists m \in N_S$, $n \in INS(m)$.
- if $n, n' \in N_S$ and $P(n) \equiv P(n')$ then $n = n'$.

Lemma 1. *Considering an XML object X and its related Structural Summary S_X, if $n_1, n_2 \in N_X$ and n_1 is ancestor of n_2, then their related nodes in S_X also have the same relationship.*

Proof. n_1 is ancestor of n_2, thus $PS(n_2) = PS(n_1) + rest$. If $s_1, s_2 \in N_S$, $n_1 \in INS(s_1)$ and $n_2 \in INS(s_2)$, then $P(n_1) \equiv P(s_1)$ and $P(n_2) \equiv P(s_2)$. In consequence, $PS(s_2) = PS(s_1) + rest$ and, because each path in S is unique, s_1 is also the ancestor of s_2. □

Use and maintenance of a *Structural Summary* should only marginally burden the query evaluation process. Read access to a *Structural Summary* is very fast, because it is a small and, typically, memory-resident object. If a new node or path instance is inserted into the document, we have to check whether the corresponding path class is present in the *Structural Summary*; otherwise, it has to be added. Such updates mostly occur when a new document is built, which means that most maintenance costs incur during document creation.[6] Hence, later document updates only require changes of the *Structural Summary* in exceptional cases, because they usually do not create new path classes.

Definition 10. A *QueryGuide* is a data structure which is defined by the tuple (S_X, RF) where S_X is a *Structural Summary* defined over an XTS object X and RF is a reference function which returns a list of DeweyIDs associated to the class nodes of $S_X: RF(c) = \{d | \exists n \in N_X, \exists m \in N_S, I(n) = (c, d), I(m) = c\}$.

Example 5. Fig. 5b shows the reference section of a *QueryGuide* which is used to implement the RF function. $RF(6) = \{1.5.1, 1.5.3, 1.7.1, 1.11.1\}$ is the set of DeweyIDs of the nodes m_1, m_2, m_3, and m_5, respectively (see Fig. 4).

4. S^3: the proposed QTP processing method

In this section, we attempt to demonstrate how the previously introduced concepts, DeweyIDs and *QueryGuide*, can be combined to provide an enhanced QTP processing method called S^3. To the best of our knowledge, S^3 is the first QTP processing method that exploits structural information of XML documents prior to performing QTP matching. Our method has two main steps. In the first step, the QTP is executed against the *Structural Summary* of the document which leads to a set of MPs (*Match Patterns*). The above set (henceforth referred to as SMP) is used as an execution plan to provide focused document

[6] Indeed, the CID numbering in Fig. 5a is obtained, if the *Structural Summary* is derived during the node-wise creation of X_1 (see Fig. 4) in document order.

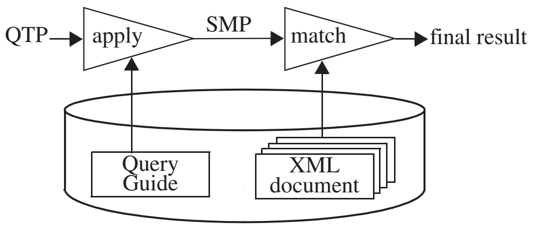

Fig. 6. Overview of the S^3 method.

access and facilitate the matching process in the second phase of S^3. Fig. 6 illustrates the interplay of the components when the S^3 method is applied.

4.1. Execution plan extraction

Processing document nodes without considering their position in the document not only leads to more costly I/Os, but also increases the number of nodes whose processing is useless and avoidable. This point motivates us to propose a solution which reduces document accesses as much as possible. In order to achieve this goal, we have to formulate the following two strategies:

1. Processing a QTP by only accessing the potential target nodes of its leaves. This is possible by using the pair of (CID, DeweyID) for the labeling of document nodes (more details in Section 4.2).
2. Providing focused document access by executing a QTP against the *Structural Summary* of the related document.

The idea behind executing QTPs against a *Structural Summary* is to find the subset of potential target nodes (of QTP leaves) whose participation in the final result definitely needs closer inspection. In other words, we attempt to remove those elements from the matching process that are not needed for the final result. These nodes are categorized into two groups: nodes not satisfying any leg of the given QTP and nodes satisfying one of the QTP legs, but without a related node matching the QTP together.

For example, consider QTP Q_1 (Fig. 1) and XTS object X_1 (Fig. 4). To evaluate Q_1, all previously introduced QTP processing methods have to process all members of PTN(D), although d_1 has no a element as an ancestor and, therefore, cannot satisfy the D-leg (//A/C/D) of Q_1. Furthermore, d_2 cannot satisfy the D-leg, because c_3 is not a child of a_2. On the other hand, all members of PTN(B) = $\{b_1,b_2,b_3,b_4\}$ can satisfy the B-leg indeed, but the *Structural Summary* S_1 (Fig. 5a) reveals that b elements with CID 13 have no d elements as counterparts to match Q_1. Thus, we can exclude b elements with CID 13 (e.g., b_3) from further processing in the matching phase. Moreover, a closer look at X_1 (Fig. 4) indicates that b_4 has not any counterpart element d, too, and b_1 has d_2 as its counterpart, but d_2 cannot satisfy the D-leg of Q_1 to produce a match. This observations about b elements with CID 7 and b elements with CID 11 cannot be obtained by only using the *Structural Summary* S_1, because B_7 and D_{11} indeed match Q_1 but sufficient information in this step is missing about nodes which are referenced by B_7 and D_{11}. Thus, these nodes should be processed in the matching phase. Execution of Q_1 against S_1 results in a single match (A_4,B_7,C_5,D_{11}) and confirms that we can exclude b elements with CID 3 or 9 and also b elements with CID 13 from any further processing steps.

Hence, to provide optimized document access in S^3, prior to processing a query on a given document, the QTP is executed against the *Structural Summary* of this document. The execution result is organized as a structure called SMP.

Definition 11. Considering an XTS object X and a QTP QTP, $n \in N_X$ matches $q \in Q_{QTP} (n \leftrightarrow q)$, if n could satisfy the related path expression from the QTP root to the query node q.

Definition 12. Considering an XTS object X and a QTP QTP, the tuple $M(n_1,n_2,\ldots,n_c)$ matches QTP($M \leftrightarrow$ QTP), if:

- $|Q_{QTP}| = c$.
- $n_i \in N_X$, $1 \leqslant i \leqslant c$.
- $\forall q_i \in Q_{QTP}$, $1 \leqslant i \leqslant c$, $n_i \leftrightarrow q_i$.
- $\forall q_i,q_j,q_k \in Q_{QTP}$, if q_k is a common ancestor of q_i and q_j, then n_k is also a common ancestor of n_i and n_j.

Q2 **Definition 13.** Considering an XTS object X and a QTP *QTP*, execution of QTP against X results in a so-called *Query Result* $QR = \{m_1,m_2,\ldots,m_r\}$:

- $|m_i| = |Q_{QTP}|$, $1 \leqslant i \leqslant r$.
- $\forall m_i \in QR$, $1 \leqslant i \leqslant r$, $m_i \leftrightarrow$ QTP.

```
procedure S^3(Q as QTP, Doc as XTS)
 1:  let SMP be the execution result of Q
     against the structural summary of Doc;
 2:  for each MP_i ∈ SMP do
 3:    matcher[i] = createMatcher(Q, MP_i, Doc);
 4:  end for
 5:  while (true) do
 6:    min = nextMatch();
 7:    if (min = null)
 8:      break;
 9:    else
10:      complete min w.r.t. its related MP and output it;
11:  end while;

function nextMatch()
12:  if no matcher is available return null;
13:  min = the minimum matcher[i].head
14:  minIndex = index of the minimum matcher[i].head;
15:  matcher[minIndex].getNext();
16:  if (matcher[minIndex].head = null)
17:    remove matcher[i];
18:  return min;

function createMatcher(Q as QTP, MP as MP, Doc as XTS)
19:  let lf be list of Q's leaves obtained by an in-order walk;
20:  ls = stream(MP, lf[1], Doc);
21:  rs = stream(MP, lf[2], Doc);
22:  let jpl be the level of MP(Q.NCA(lf[0], lf[1]));
23:  matcher = new QTPMatcher(ls, rs , jpl);
24:  for i = 3 to size of lf do
25:    rs = stream(MP, lf[i], Doc);
26:    let jpl be the level of MP(Q.NCA(lf[i-1], lf[i]));
27:    matcher = new QTPMatcher(matcher, rs , jpl);
28:  end for
29:  return matcher;

function stream(MP as MP, lf as QTPNode, Doc as XTS)
30:  let QG be the QueryGuide of Doc;
31:  let cid be the CID of that MP's member which is related to lf;
32:  return new NodeStream(QG, cid);
```

Fig. 7. Pseudo-code of the S^3 algorithm.

We refer to the Query Result as SMP if it is derived from the *Structural Summary* of a document. In this case, each member of the SMP is also called MP. It is worth noting that, in this step, any QTP processing method could be used to execute the given QTP against the *Structural Summary* which is a small object and, therefore, SMP construction has only insignificant cost. Based on the *Structural Summary* definition, we can claim that no potential final match is discarded by the execution of QTP against the document's *Structural Summary*, if the matching process uses the resulting SMP as its input. In other words:

Theorem 1. *For each final match of a given query tree pattern* QTP *against an XML object X with Structural Summary S, exactly one* MP *could be found that has the same sequence of CIDs as the sequence of CIDs of that match.*

Proof. Consider a final match $M(n_1,n_2,\ldots,n_m)$ and $CM(c_1,c_2,\ldots,c_m)$ such that $M \leftrightarrow$ QTP and $c_i = \mathrm{CID}(n_i)$. As a consequence, we could construct the tuple $\mathrm{MP}(s_1,s_2,\ldots,s_m)$ such that $s_i \in N_S$, $1 \leqslant i \leqslant m$ and $c_i = \mathrm{CID}(s_i)$. It is clear that $\mathrm{CID}(n_i) = \mathrm{CID}(s_i)$ and, based on Definition 8, we can conclude that $n_i \in INS(s_i)$ and $P(n_i) \equiv P(s_i)$. Because $PS(n_i)$ match the $q_i \in Q_{\mathrm{QTP}}$, $P(s_i)$ can match q_i, too. Thus, we can derive that $s_i \leftrightarrow q_i$. On the other hand, consider that $q_i, q_j, q_k \in Q_{\mathrm{QTP}}$, q_k is a common ancestor for q_i and q_j, because MP $\leftrightarrow$ QTP, then n_k is also a common ancestor for n_i and n_j and based on Lemma 1, s_k is also a common ancestor for s_i and s_j. Hence, all criteria of Definition 12 are met and we can conclude that MP $\leftrightarrow$ QTP. It is straightforward to see that if another match pattern MP′ is found then MP = MP′. □

4.2. Matching process

The matching process is fed by the SMP which is created in the first step of the algorithm. Each MP of the SMP is used to produce a subset of the final matches, precisely those matches whose CIDs match the selected MP. Theorem 1 demonstrates that it is possible to classify the final matches of a given QTP into some categories and each category would belong to one of the MPs in the SMP. As a consequence, each category of results could be produced by extracting only those nodes from the reference section of the *QueryGuide* which have the same CIDs as the *Structural Summary* nodes of the related MP.[7] It is worth noting that it is not necessary to extract all of these nodes, because we can derive label and name of all ancestors of any node using its label together with the *Structural Summary* (Section 3.2). Thus, the matching process could be performed by only extracting those nodes which are related to the leaves of the given QTP. This means that, during the entire matching process, we only need to access the selected subset of nodes which are related to the leaves of the QTP.

Fig. 7 depicts the pseudo-code of S^3 algorithm. Execution of a given QTP Q against the *Structural Summary* of an XML document *Doc* results in an SMP object (line 1), and after that a *QTPMatcher* object is created for each MP of the resulting SMP by use of procedure *createMatcher* (lines 2–4). In fact, each returned *QTPMatcher* is a chain of *QTPMatcher* objects. Using function *createMatcher*, first an ordered list of QTP leaves is created by a pre-order walk through Q (as a result, the order of the extracted leaves is the same as the left-to-right order of leaves when Q is printed as a tree). Then for the first two leaves of Q, a *QTPMatcher* is constructed (lines 20–23). For each leaf ($lf[1], lf[2]$), a *NodeStream* object (see lines 1–4 in Fig. 8) is created

[7] Henceforth, we refer to a member of a given MP MP, which is related to a given QTP node q, as MP(q).

```
class NodeStream implements InputStream
  constructor(QG as QueryGuide, cid as CID)
  1:  let stream be a stream of sorted DeweyIDs of QG.RF(cid)
  2:  let this.head be the current node of stream

   procedure getNext()
  3:  advance stream to the next node;
  4:  let head be the current node of stream;

class QTPMatcher implements InputStream
  constructor(ls, rs as MatcherInputStream, jpl as integer)
  5:  let this.resultQueue be an empty queue of matches
  6:  this.jpl = jpl;
  7:  this.ls = ls;
  8:  this.rs = rs;
  9.  let this.lastJoinedNode be the QTP node related to rs;

   procedure getNext()
 10:  if (!resultQueue.empty)
 11:     head = resultQueue.dequeue();
 12:     return;
 13:  end if;
 14:  lKey = getLast(ls.head);
 15:  lList = advance(ls);
 16:  rKey = rs.head;
 17:  rList = advance(rs);
 18:  while(!ls.finished() and !rs.finished())
 19:     if (lKey.prefix(jpl) = rKey.prefix(jpl))
 20:        for each combination of lList and rList produce a match,
            then sort them w.r.t. this.lastJoinedNode and add them
            to the resultQueue;
 21:        break;
 22:     elseif (lKey.prefix(jpl) < rKey.prefix(jpl))
 23:        lKey = getLast(ls.head);
 24:        lList = advance(ls);
 25:     else
 26:        rKey = rs.head;
 27:        rList = advance(rs);
 28:     endif
 29:  end while
 30:  head = resultQueue.dequeue();

   function advance(s as MatcherInputStream)
 31:  let list be an empty list of matches
 32:  list.add(s.head);
 33:  s.getNext();
 34:  while (!s.finished and list.head.prefix(jpl) = s.head.prefix(jpl))
 35:     list.add(s.head);
 36:     s.getNext();
 37:  end while;

   function getLast()
 38:  return DeweyID in head which belongs to lastJoinedNode
```

Fig. 8. Pseudo-code of the *NodeStream* and *QTPMatcher* classes.

which maintains a stream of sorted DeweyIDs which are referenced in the Reference Section of the *QueryGuide* by the CID of that MP's member which is related to the given QTP leaf (MP($lf[1]$),MP($lf[2]$)). The last important parameter to construct the *QTPMatcher* is the join point level of the above QTP leaves which is used in the matching process (procedure *getNext* in Fig. 8) to satisfy the last criterion of Definition 12. In order to show that *QTPMatcher* works correctly, we now focus on QTPs having only two leaves.

Lemma 2. *Outputs of procedure getNext of class QTPMatcher (see* Fig. 8*) are candidates to produce matches related to the associated MP of QTPMatcher for QTPs having only two leaves.*

Proof. In order to find the matches of a given QTP Q with two leaves (l_1, l_2) based on a given MP MP, the related DeweyIDs of *ls* and *rs* should be compared. The pair (d_1, d_2) is a candidate to produce a match, if d_1 and d_2 have proper ancestors to match Q. As depicted in function stream (see lines 30–32 in Fig. 7), *ls* is a stream of nodes having the same CID as that of MP(l_1). Thus, all nodes of *ls* including d_1 match the leg of Q which is related to l_1. The same story is also true for d_2 and l_2. The remaining point w.r.t. Definition 12 is to prove that d_1 and d_2 have common ancestors that match the common ancestors of l_1 and l_2.

Assume that *jp* is the nearest common ancestor (NCA) of l_1 and l_2, then *jpl* is the level of MP(*jp*). With respect to lines 19–20 in Fig. 8, d_1 and d_2 have a common ancestor c at the *jpl* level of the document. It is clear that c matches query node *jp*. It is also straightforward to show that all other common ancestors of l_1 and l_2 have a common match in the document, which is one of the nodes in the path from the document root to node c. As a consequence, the outputs of procedure *getNext* can be extended to full matches which are able to satisfy Definition 12. □

Theorem 2. *Procedure getNext of class QTPMatcher (see* Fig. 8*) computes all possible matches related to the associated MP of QTPMatcher for QTPs having only two leaves.*

Proof. Based on Lemma 2, consider that the pair (d_1, d_2) is a candidate to produce a full match. Also assume that (d_1, d_3) and (d_4, d_2) are two other candidates. With respect to line 19, d_1, d_2, d_3, d_4 have a common ancestor at the *jpl* level of the document. Thus, (d_4, d_3) is also a candidate. We can derive from this simple example that all node combinations of *ls* and *rs* having DeweyIDs with the same prefix up to the *jpl* level are candidates for producing complete matches. A closer look at procedure *getNext* shows that, if the sets of nodes having DeweyIDs with the same prefix up to the *jpl* level are replaced by their prefixes (see function *advance*), then *getNext* would be a simple merge-join algorithm for two sorted sets, which in conjunction with line 20 can produce all required matches. □

Example 6. Consider QTP Q_1 (Fig. 1) and XTS object X_1 (Fig. 2). Execution of Q_1 against S_1, as the *Structural Summary* of X_1 (see Fig. 5), results in a single MP(A_4, B_7, C_5, D_{11}). Thus, only one *QTPMatcher* is needed to be created with {1.5.1.1, 1.7.1.1, 1.11.1.1} as its left stream related to B_7, {1.7.3.1.1} as its right stream related to D_{11} and A_4 as the join point. Since A_4 is in the second level of S_1, the extracted DeweyIDs should be compared based on their prefix of length 2. As a consequence, it is clear that the only candidate in this example is (1.7.1.1, 1.7.3.1.1). It is not difficult to complete this candidate to form a full match. Because *A* and *C* are ancestors of *B* in Q_1 and A_4 and C_5 are at the second and third level of S_1, the full match could be formed as following: (1.7, 1.7.1.1, 1.7.3, 1.7.3.1.1).

The matching process for QTPs having more than two leaves is the same as introduced for QTPs having two leaves, but it is performed recursively. For the first two leaves of a given QTP, a *QTPMatcher*, assume *matcher*$_1$, is created (lines 20–23 in Fig. 7). Then for the third leaf, a new *QTPMatcher*, assume *matcher*$_2$, is created based on *matcher*$_1$ as the left input and the *NodeStream* object of the third leaf, assume *ns*$_3$ as the right input (lines 24–28 in Fig. 7). *matcher*$_2$ joins partial matches of *matcher*$_1$ with nodes which are maintained by *ns*$_3$. The determination of the join point and node comparisons in *QTPMatchers* like *matcher*$_2$ are based on the last query node which has been added to the left input of *QTPMatcher* (here, *matcher*$_1$), as characterized by the use of function *getLast* (lines 14 and 23 in Fig. 8).

Lemma 3. *Consider l_m, l_n, l_k be three leaves of a given QTP Q, and $l_m < l_n < l_k$ holds for a pre-order traversal through Q. Then NCA(l_m, l_k)*[8] *is ancestor or self of NCA(l_m, l_n) and NCA(l_n, l_k).*

Proof. NCA(l_m, l_k) and NCA(l_m, l_n) are both ancestors of l_m. As a result, one of them is ancestor or self of the other. If NCA(l_m, l_n) is ancestor of NCA(l_m, l_k), then l_n would be visited before or after NCA(l_m, l_k) and all descendants of NCA(l_m, l_k) (obvious property of a pre-order traversal). Thus, l_n would be visited before l_m or after l_k and both of them are in contradiction to the assumptions of Lemma 3. □

Lemma 4. *Outputs of procedure getNext of class QTPMatcher (see Fig. 8) are candidates to produce matches related to the associated MP of QTPMatcher.*

Proof. Lemma 2 is a special case of this lemma for QTPs having two leaves. For QTPs with more than two leaves, assume that MP is the associated MP of a *QTPMatcher* and $l_1, l_2, \ldots, l_m$ be the leaves of the given QTP Q, and $l_1 < l_2 < \cdots < l_m$ holds for a pre-order traversal through Q. Assume $(d_1, d_2, \ldots, d_n)$ is a candidate w.r.t. $(l_1, l_2, \ldots, l_n)$. If there exists a d_{n+1} which satisfies line 19 in Fig. 8, then the following two facts can be derived:

- Regarding the above assumption, d_n and d_{n+1} have the same prefixes w.r.t. MP NCA(l_n, l_{n+1}). Therefore, w.r.t. Lemma 2 (d_n, d_{n+1}) is a candidate for (l_n, l_{n+1}). On the other hand, for each $i < n$, NCA(l_n, l_{n+1}) is a descendant or self of NCA(l_i, l_{n+1}) (Lemma 3) and, therefore, MP(NCA(l_i, l_{n+1})) is self of MP(NCA(l_n, l_{n+1})) or it is placed at higher levels of the corresponding *Structural Summary*. As a consequence, because d_n and d_{n+1} have the same prefixes w.r.t. MP(NCA(l_n, l_{n+1})), they also have the same prefixes w.r.t. MP(NCA(l_i, l_{n+1})) for each $i < n$.
- Since $(d_1, d_2, \ldots, d_n)$ is a candidate for $(l_1, l_2, \ldots, l_n)$, for each $i < n$, d_i and d_n have the same prefixes w.r.t. MP(NCA(l_i, l_n)). On the other hand, w.r.t. Lemma 3, for each $i < n$, NCA(l_i, l_{n+1}) is an ancestor or self of NCA(l_i, l_n) and, therefore, MP(NCA(l_i, l_{n+1})) is self of MP(NCA(l_i, l_n)) or it is placed at higher levels of the corresponding *Structural Summary*. As a result, d_i and d_n have the same prefixes w.r.t. MP(NCA(l_i, l_{n+1})).

Based on the above facts, we can conclude that, for each $i \leqslant n$, d_i and d_{n+1} have the same prefixes w.r.t. MP(NCA(l_i, l_{n+1})). Thus, $(d_1, d_2, \ldots, d_n, d_{n+1})$ satisfies Definition 12 and is a candidate to produce a match. □

Theorem 3. *Procedure getNext of class QTPMatcher (see Fig. 8) computes all possible matches related to the associated MP of QTPMatcher.*

Proof. Theorem 2 is a special case of this theorem for QTPs having two leaves. Now for QTPs with more than two leaves, assume that MP is the associated MP of the *QTPMatcher* and $l_1, l_2, \ldots, l_m$ be the leaves of the given QTP Q, and $l_1 < l_2 < \cdots < l_m$ holds for a pre-order traversal through Q. The skeleton of procedure *getNext* is a merge-join algorithm for two sorted sets. Procedure *getNext* is recursive and joins the related stream of l_{i+1} (*rs*) to the existing results (*ls*) in the ith recursion. *rs* is always a sorted stream of nodes (*NodeStream*). *ls* is a stream of partial matches which are sorted based on their last joined node (l_i) (see line 20 in Fig. 8). As a result, it is possible to group the members of *ls* and *rs* into sets with members having the same prefix up to their *jpl* level (level of MPNCA(l_i, l_{i+1})). Thus, in conjunction with the Cartesian product described in line 20, all matches related to $(l_1, l_2, \ldots, l_{i+1})$ are produced in the ith recursion of procedure *getNext*. □

Example 7. Consider *Structural Summary* S_2 and QTP Q_3 in Fig. 9. Execution of Q_3 against S_2 results in a single match MP(E_2, N_3, G_4, H_5, L_8). *G*, *H*, and *L* are three leaves of Q_3. Hence, three streams of elements corresponding to G_4, H_5, and L_6 have to be created. Assume that *RF*(4) = {1.3.5.3, 1.5.7.1, 1.5.7.5, 1.9.5.1}, *RF*(5) = {1.1.3.5, 1.5.7.3, 1.5.7.7, 1.7.9.11, 1.9.5.3}, and *RF*(8) = {1.3.7, 1.5.11, 1.7.3, 1.7.5, 1.9.3}. The relevant matching process is depicted in Fig. 10, which has two main steps. In

[8] Henceforth, we refer to the nearest common ancestor of two given nodes n_1, n_2 in a tree as NCA(n_1, n_2).

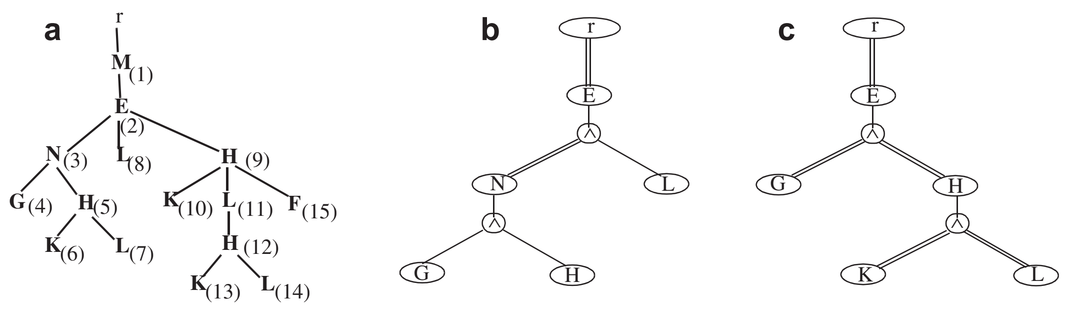

Fig. 9. (a) A sample *Structural Summary* (S_2); (b) QTP Q_3; (c) QTP Q_4.

the first step, two streams related to G and H are joined with each other. The NCA of G and H is N. Therefore, g and h elements can be joined if they have a common matching ancestor n. N_3 is at the third level of S_2. As a result, if labels of a pair of g and h elements have the same prefix with length 3, they have a common n ancestor and can be joined with each other. It is clear that this pair also has a common e ancestor. In the second step of the match, l elements are joined with pairs of g and h. E_2 (related node to NCA of H and L) is at the second level of S_2. Therefore, an l element can be joined with a pair of g and h, if the labels of l and g elements have the same prefix with length 2. This means that they have a common e ancestor. It is straightforward that the above l element also has the same common e ancestor with the g element (see Lemma 4).

As illustrated in Fig. 10, the matching process is done by a pipelining strategy (see line 20 in Fig. 8). Each time a join is performed, its results are sent to the next step. Therefore, there is no need to keep the entire intermediate results of each step up to the end of that step. Intermediate results are discarded as soon as they are processed in the next join step. As a consequence, the matching process is done without consuming a huge amount of memory for storing the intermediate results.

4.3. Optimized S^3

In the previous sections, we have shown how S^3 uses a *QueryGuide* to provide focused search as a key factor for I/O reduction. However, S^3 deviates from its goals in some cases. The problem arises when the execution of a given QTP results in an SMP which contains nodes repeatedly occurring in its diverse MPs. This means that some elements have to be accessed more than once:

Example 8. Consider QTP Q_4 and *Structural Summary* S_2 in Fig. 9. Execution of Q_4 against S_2 results in the following SMP: $\{mp_1(E_2,G_4,H_5,K_6,L_7)$, $mp_2(E_2,G_4,H_9,K_{10},L_{11})$, $mp_3(E_2,G_4,H_{12},K_{13},L_{14})$, $mp_4(E_2,G_4,H_9,K_{10},L_{14})$, $mp_5(E_2,G_4,H_9,K_{13},L_{11})$, $mp_6(E_2,G_4,H_9,K_{13},L_{14})\}$. For each MP, a separate *QTPMatcher* has to be instantiated and, as a consequence, elements which are related to G_4 (those having CID 4) have to be fetched six times.

To avoid these additional and unnecessary I/Os, we propose an optimized version of our QTP processing method, called OS^3. As depicted in Fig. 11, the idea behind OS^3 is to assign MPs having the same nodes to some grouped MPs, called GMPs. Each GMP is responsible to produce all matches related to its wrapped MPs.

In OS^3, after a QTP is executed against a *Structural Summary*, the resulting SMP is transformed to a grouped SMP (GSMP) by the calling function *groupSMP* (line 2). In order to classify MPs by the function *groupSMP* (lines 32–37), the number of distinct occurrences of its related nodes in the SMP is counted for each QTP leaf. That leaf having the minimum distinct occurrences

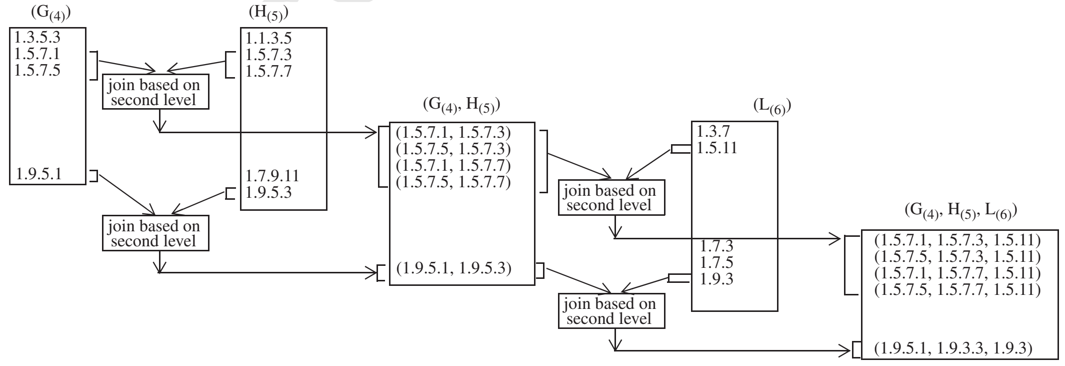

Fig. 10. Matching process for Q_3.

```
procedure OS$^3$($Q$ as QTP, $Doc$ as XTS)
 1:  let $SMP$ be the execution result of $Q$ against
     the structural summary of $Doc$;
 2:  $SGMP$ = groupSMP($Q$, $SMP$);
 3:  for each $GMP_i \in SGMP$ do
 4:    $matcher[i]$ = createGroupMatcher($Q$, $GMP_i$, $Doc$);
 5:  end for
 6:  let $resultQueue$ be an empty queue of matches
 7:  while (true) do
 8:    $match$ = nextMatch($resultQueue$);
 9:    if ($match$ = null)
10:      break;
11:    else
12:      Output $match$;
13:  end while;

function nextMatch($resultQueue$ as Queue)
14:  if (not $resultQueue$.empty)
15:    return $resultQueue$.dequeue();
16:  if no matcher is available return null;
17:  $min$ = min{$matcher[i].head$};
18:  $minIndex$ = minarg$_i${$matcher[i].head$};
19:  $matcher[minIndex]$.getNext();
20:  if ($matcher[minIndex].head$ = null)
21:    remove $matcher[i]$;
22:  $found$ = false;
23:  for each $MP_j \in GMP_{minIndex}$ do
24:    if ($min$ matches $MP_j$)
25:      complete $min$ w.r.t. $MP_j$ to be a full match for $Q$
         and add it to the $resultQueue$
26:      $found$ = true;
27:    end if;
28:  end for;
29:  if (not $found$)
30:    return nextMatch($resultQueue$);
31:  return $resultQueue$.dequeue();

function groupSMP($Q$ as QTP, $SMP$ as SMP)
32:  let $lf$ be an array of $Q$'s leaves
33:  let $lfc_i$ be distinct number of different CIDs related to leaf $lf_i$
34:  let $maxLeaf$ be the $lf_{max}$ such that $lfc_{max}$ be the minmum of $lfc$
35:  classify $MP_j \in SMP$ into $lfc_{max}$ groups ($GMP_k$, $1 \le k \le lfc_{max}$)
     such that in each group nodes related to $maxLeaf$ be the same
36:  let $SGMP$ be array of $GMP_k$, $1 \le k \le lfc_{max}$
37:  return $SGMP$

function createGroupMatcher($Q$ as QTP, $GMP$ as GMP,
         $Doc$ as XTS)
38:  let $lf$ be list of $Q$'s leaves obtained by in-order walking;
39:  $ls$ = stream($GMP$, $lf[1]$, $Doc$);
40:  $rs$ = stream($GMP$, $lf[2]$, $Doc$);
41:  $jpl$ = highestJoinPint($lf[1]$, $lf[2]$, $GMP$);
42:  $matcher$ = new QTPMatcher($ls$, $rs$, $jpl$);
43:  for $i$ = 3 to size of $lf$ do
44:    $rs$ = stream($GMP$, $lf[i]$, $Doc$);
45:    $jpl$ = highestJoinPint($lf[i-1]$, $lf[i]$, $GMP$);
46:    $matcher$ = new QTPMatcher($matcher$, $rs$, $jpl$);
47:  end for
48:  return $matcher$;

function highestJoinPoint($lf1$, $lf2$ as QTPNode, $GMP$ as GMP)
49:  for each $MP_i \in GMP$ do
50:    let $jp_i$ be the level of $MP(Q$.NCA($lf1$, $lf2$));
51:  end for;
52:  return min{ $jp_i$};

function stream($GMP$ as GMP, $lf$ as QTPNode, $Doc$ as XTS)
53:  let $QG$ be the QueryGuide of $Doc$;
54:  let $cids$ be set of CIDs related to $lf$ in $GMP$;
55:  return new GroupedNodeStream($QG$, $cids$);
```

Fig. 11. Pseudo-code of the OS^3 algorithm.

has the maximum repeated occurrences in the SMP. Therefore, MPs which contain the same nodes w.r.t. the above leaf are grouped with each other into a separate GMP.

After the grouping of MPs, a separate *QTPMatcher* is instantiated for each GMP (lines 3–4). *QTPMatchers* used for OS^3 are the same as those applied in S^3. But the input streams feeding the *QTPMatchers* and the way to determine join points are different in OS^3. Associated to each leaf in a GMP, more than one CID may exist. For each CID, a (sorted) *NodeStream* is therefore instantiated. Then, a *GroupedNodeStream* object is used to merge the resulting streams to a single sorted stream of elements (see Fig. 12). Furthermore, the determination mechanism for join points in S^3 is not applicable to OS^3, because the wrapped MPs in a GMP have not necessarily the same join points. For a given GMP, the join point of each pair of QTP leaves is the join point of a wrapped MP which has the highest level in the *Structural Summary* among other related join points (see function *highestJoinPoint* in Fig. 11).

False positives did not occur during node matching in an S^3 evaluation. In OS^3, however, these algorithmic changes may cause false positives. *QTPMatcher* may produce results consisting of CIDs which match one or more MPs, but do not produce a full match. This means that, while each element in these results matches its associated QTP leaf, there are at least two leaves (e.g., lf_i and lf_j) for which the related elements have no common ancestor to match NCA of lf_i and lf_j. This problem arises when the common ancestor of two elements is searched by *QTPMatcher* at a level which is probably higher than the actually required level. Furthermore, because *NodeStreams* maintain elements with different CIDs, *QTPMatcher* may produce results related to none of the MPs. Therefore, the *QTPMatcher* output has to be checked against the related MPs of *QTPMatcher* (lines 22–30 in Fig. 11). It is worth noticing that the output of a *QTPMatcher* may match more than one wrapped MP of a given QTP and, hence, more than one full match will be produced for them. This happens to MPs when their members have the same CIDs for QTP leaves.

Example 9. Again consider QTP Q_4 and *Structural Summary* S_2 in Fig. 9. As described in the previous example, G_4 has the maximum occurrence in the corresponding SMP. As a result, the SMPs have to be grouped based on leaf G. Because all five MPs have the same node related to leaf G (G_4), the resulting SGMP only has one GMP $\{(E_2, G_4, (H_5, H_9, H_{12}), (K_6, K_{10}, K_{13}), (L_7, L_{11}, L_{14}))\}$. This means that OS^3 fetches elements with CID 4 only once instead of six times done by S^3. The join point

```
class GroupedNodeStream implements InputStream
  constructor($QG$ as QueryGuide, $cids$ as set of CID)
   1:  for each $cid_i \in cids$ do
   2:      $str_i$ = new NodeStream($QG$, $cid_i$);
   3:    getNext();

  procedure getNext()
   4:  if (no stream is available)
   5:      this.$head$ = null;
   6:      return;
   7:  end if;
   8:  min = minarg$_i$($str_i.head$);
   9:  this.$head$ = $str_{min}.head$;
  10:  $str_{min}$.getNext();
  11:  if ($str_{min}.head$ = null) remove $str_{min}$;
```

Fig. 12. Pseudo-code of the *GroupedNodeStream* class.

level for leaves G and K is set to 2, because this level is also 2 for all MPs. The minimum join point level for leaves K and L is 3. (The join point level for these leaves is 3 for mp_2, mp_4, mp_5, and mp_6. It is 4 for mp_1 and 5 for mp_3.) Now assume that $RF(4) = \{1.3.5.3, 1.9.5.1\}$, $RF(6) = \{1.3.7.1.3\}$, $RF(7) = \{1.3.7.7.3\}$, $RF(10) = \{1.5.3.7\}$, $RF(11) = \{1.3.7.9\}$, $RF(13) = \{1.9.3.5.9.1\}$, and $RF(14) = \{1.9.3.5.9.3\}$. For the above GMP, *QTPMatcher* returns combinations of three elements w.r.t. the leaves of QTP (G, K, L) as follows: $\{m_1[(4, 1.3.5.3), (6, 1.3.7.1.3), (7, 1.3.7.7.3)], \; m_2[(4, 1.3.5.3), (6, 1.3.7.1.3), (11, 1.3.7.9)], \; m_3[(4, 1.9.5.1), (13, 1.9.3.1.5.9.1), (14, 1.9.3.5.9.3)]\}$. These results have to be checked to remove false positives. m_1 has proper CIDs to match mp_1, but m_1 cannot match mp_1, because 1.3.7.1.3 and 1.3.7.7.3 have the same prefix with length 3, but the actual join point level for leaves K and L in mp_1 is 4 (level of H_5). The next output of *QTPMatcher* is m_2 which cannot match any MPs. The problem with m_2 is that, while 1.3.7.1.3 and 1.3.7.9 belong to CIDs which cannot match any MPs, they incidentally have the same prefix with length 3 and, therefore, they are joined by *QTPMatcher*. The last output of *QTPMatcher* is m_3 which matches both m_3 and m_6.

Theorem 4. *The method OS^3 correctly computes all possible matches for a given QTP Q and an XML document Doc.*

Proof. By transforming an SMP to an SGMP, all corresponding MPs are included and no MP is deleted. Therefore, we do not lose any class of final matches. The related *QTPMatcher* of a GMP is fed by a sorted stream of elements which include all CIDs of the corresponding MPs of that GMP. Furthermore, the join point level for each pair of leaves is set to the highest level among the corresponding MPs. As a result, *QTPMatcher* returns all required combinations of elements and some false positives, which are checked and removed (see lines 22–30 in Fig. 11). □

5. Experimental results

5.1. Experimental setup

Using our native, Java-based XML database system, called XTC [24], we have implemented the QTP processing methods S^3 and OS^3 as well as *TwigStack* [4], *TJFast* [19], *TwigList* [22] and its external version *E-TwigList* [22]. The system configuration for all performance experiments was setup under Java 1.6.0_03 on a 2x3.2 GHz Pentium IV computer, 1 GB main memory, 2x80 GB hard disks, running GNU/Linux, where the maximum heap size of the Java Virtual Machine was 800 MB.

We want to present an exhaustive performance comparison of the entire spectrum of algorithms considered (see Table 1). Because the *Structural Join* [1] is too simplistic and up to two orders of magnitude slower than the best methods present [20], we have dropped it from our cross-comparison. *TwigStack* is the first competitive method evaluated. Compared to it, *TJFast* aims to reduce I/O cost by processing only potential target nodes of QTP leaves. Moreover, *TwigList*, a refined version of *Twig²-Stack*, attempts to achieve even better performance by eliminating the merging phase needed in *TwigStack* and *TJFast*. We want to show the superiority of our proposals S^3 and OS^3, which is essentially achieved by the interplay of both concepts DeweyIDs and *QueryGuide*. To illustrate the robustness, scalability, and structure insensitivity of the methods, we have chosen a spectrum of different and well-known datasets: DBLP [17], Nasa [23], SwissProt [23], and XMark [25] with scaling factor 5, whose characteristics are shown in Table 2. Data size refers to the dataset in its plain text format, whereas the number of nodes as well as the maximum and average depth are computed from the physical representation of these datasets in XTC. For each dataset, we have specified a set of queries with different features. Single path queries are used to analyze the method behavior when path join operations are not needed. To explore tree queries having two or more leaves, some of the QTPs

Table 2
Characteristics of XML datasets used

Aspect	DBLP	Nasa	SwissProt	XMark(5)
Data size (MB)	404	23.88	109	558
Nodes (Mio)	31.88	1.22	11.4	23.96
Max/avg depth	8/4.8	10/7.7	7/5.4	14/7.8

are shallow and some are deeper. Furthermore, different combinations of *P–C* and *A–D* relationships are used in our queries. Table 3 presents the collection of queries used.

We have compared our methods in terms of *total execution time*, *I/O time*, and *number of elements read*:

- *Total execution time* is the elapsed time between the arrival of the query in XTC and the delivery of the complete result to the user.
- *I/O time* is the entire time spent to fetch elements from the document.
- *Number of elements read* indicates how many elements have to be read in a matching process.

Normally, the number of elements read is large for methods like *TwigStack* and *TwigList*, because they read all elements related to all query nodes. *TJFast* reads fewer elements, because it processes only elements related to QTP leaves, and we expect that this number is lower for our methods S^3 and especially OS^3.

5.2. Single path queries

Fig. 13 shows the results of our experiments for single path queries on the selected datasets. Because the response time range is very large, we have chosen a logarithmic scale. Most remarkably, S^3 and OS^3 are five times faster in the average than *TwigStack* and *TwigList*. To explain these results, we interpret some indicative cases. Consider, e.g., the number of elements to be read for *D1* by *TwigStack* and *TwigList*: they have to read four times more elements than S^3 which, therefore, gains a factor of 5. Compared to *TJFast*, S^3 is still more than two times (2.2) faster. Although *TJFast* and S^3 have to read only title elements for *D1*, S^3 has the advantage to access only those title elements which are children of an article element. In contrast, *TJFast* has to read all title elements in DBLP, although most of them are children of other elements. Hence, S^3 has to read only one third (0.36) compared to *TJFast*. For *D2*, *X1*, and *X2*, however, *TJFast* and S^3 obtain the same results. This kind of behavior can be explained, e.g., for *X1*, because all price elements in the XMark dataset match *X1*; hence, *TJFast* and S^3 have to read the same number of elements. For *N1*, however, S^3 and OS^3 achieve two orders of magnitude performance gain over the other methods, because *N1* is a very selective path query. Here, the competitors, especially *TwigStack* and *TwigList*, cannot take advantage of such situations, because they process the elements without considering their position in the document.

Table 3
Queries used in the experiments

Name	Query	Database	Matches
D1	//article/title	DBLP	346,554
D2	/dblp/inproceedings/booktitle	DBLP	582,679
D3	//inproceedings//title[.//i]//sub	DBLP	304
D4	/dblp/inproceedings[title]/author	DBLP	1,519,938
D5	//inproceedings[author][.//title]//booktitle	DBLP	1,519,938
D6	/dblp/inproceedings[.//cite/label][title]//author	DBLP	132,902
D7	//article[.//mdate][.//volume][.//cite]//journal	DBLP	13,785
D8	//inproceedings[.//title[//sup/i]//tt][//cite/label]//booktitle	DBLP	0
X1	/site/closed_auctions/closed_auction/price	XMark	48,756
X2	/site/regions//item/location	XMark	108,750
X3	/site//open_auction[.//bidder/personref]//reserve	XMark	146,982
X4	//people//person[.//address/zipcode]/profile/education	XMark	15,857
X5	//item[location]/description//keyword	XMark	136,260
X6	//item[location][.//mailbox/mail//emph]/description//keyword	XMark	86,568
X7	//item[location][quantity][//keyword]/name	XMark	207,639
X8	//people//person[.//address/zipcode][id]/profile[.//age]/education	XMark	7991
N1	//textFile/decription//footnote//para	Nasa	16
N2	//revisions[//year][//para]//creator	Nasa	1043
N3	//tableHead[./tableLinks/tableLink/title]//fields/field[definition]/name	Nasa	103,380
N4	//dataset[reference[//keyword][//description[para][heading]]]/subject	Nasa	370
S1	//Features/DOMAIN/Descr	SwissProt	47,234
S2	//Entry//PIR[prim_id][sec_id]	SwissProt	30,427
S3	//Entry/Features[/DISULFID[from][to]/Descr][/CHAIN[from][to]/Descr]	SwissProt	23,437
S4	//Entry[mtype][Mod][Descr]/id	SwissProt	150,000

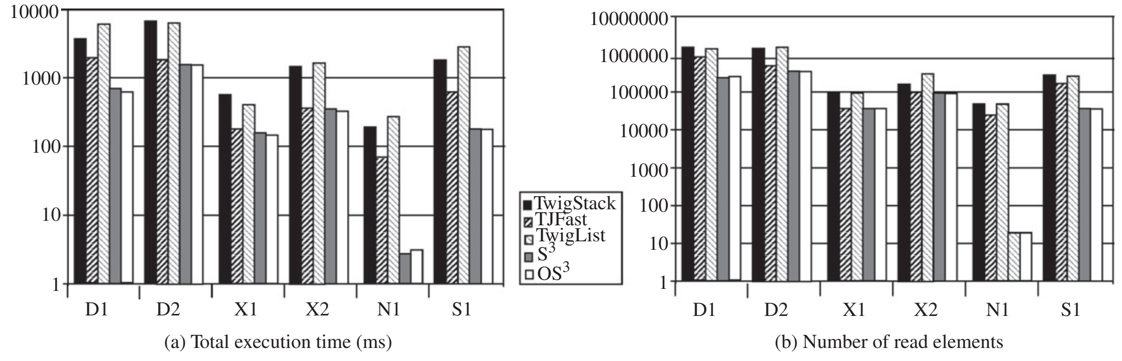

Fig. 13. Experimental results for single path queries.

5.3. Tree queries

We have continued our tests using several XML tree queries with varying patterns on the same datasets. For shallow queries, the number of query nodes for which the related elements are needed to be processed get closer for methods like *TJFast*, S^3, and OS^3 compared to *TwigStack* and *TwigList*. Moreover, some queries with three or more leaves are selected. This type of queries is more suitable for *TwigList* which does not rely on expensive merging phases.

Fig. 14a–c depicts our experimental results for tree queries on the DBLP dataset in terms of total execution time, I/O time, and number of elements read. Again, S^3 and OS^3 provide a substantial performance advantage: they are four times faster in the average than *TwigStack* and *TJFast*. Moreover, they are three times faster than *TwigList*. Our experiments for *D6*, *D7*, and *D8* illustrate the efficiency of using a *Structural Summary* in S^3 and OS^3. Although *D6* and *D7* are shallow and *D8* has many leaves, S^3 and OS^3 only have to read about half of the elements than the other methods. For *D3*, *TJFast* has a slight gain over S^3 and OS^3, because *TJFast* has lower I/O cost in this case. There are only a few number of *sub* and *i* elements in DBLP dataset which can be indexed using only a few pages, whereas S^3 and OS^3 have to access more often element indexes than *TJFast*, because they only read elements related to a single CID during each access.

Fig. 15 shows our experimental results for the XMark (scale 5) dataset. As depicted in Fig. 15a, OS^3 is three times faster than the other methods. S^3 also obtains the same performance except for *X5*, *X6*, and *X7*. Here, S^3 is about three times slower than *TJFast* for *X6* and *X7* and it is 1.3 times slower for *X5* and has the worst performance among all methods. This low performance for *X6* arises, because 162 different MPs related to this query are created by S^3, whereas these MPs are grouped into 6 GMPs in OS^3. This means that some of the elements are fetched 27 times in S^3 and also processed 27 times in 27 different *QTPMatcher* executions, while these nodes are fetched and processed only once in OS^3. As a result, processing time and I/O cost for queries like *X6* (see Fig. 15a and b) are very high; OS^3 can reduce these costs by grouping related MPs into a single GMP. Also, S^3 creates 54 distinct MPs for *X7* and 72 distinct MPs for *X7*. These MPs are grouped in 6 GMPs in OS^3. As a consequence, OS^3 is about more than two times faster than *TJFast* for *X5*, *X6*, and *X7*.

Fig. 16 shows our experimental results for Nasa and SwissProt. As depicted in Fig. 16a, S^3 and OS^3 are, in the average, six times faster than *TJFast* and eight times faster than *TwigStack* and *TwigList*. For *N4* and *S2*, we achieve with S^3 and OS^3 about one order of magnitude performance gain over the other methods. Fig. 16b and c also reflects this behavior in terms of I/O cost and elements processed for these queries.

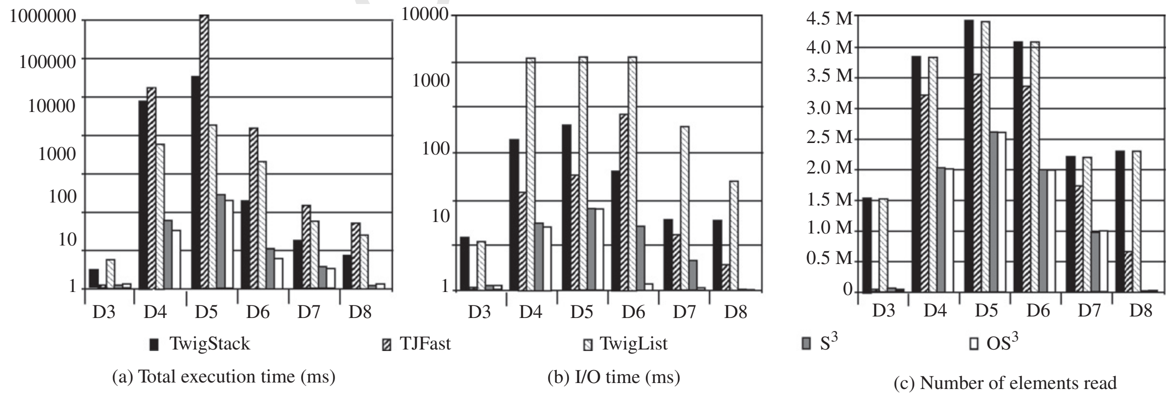

Fig. 14. Experimental results for DBLP.

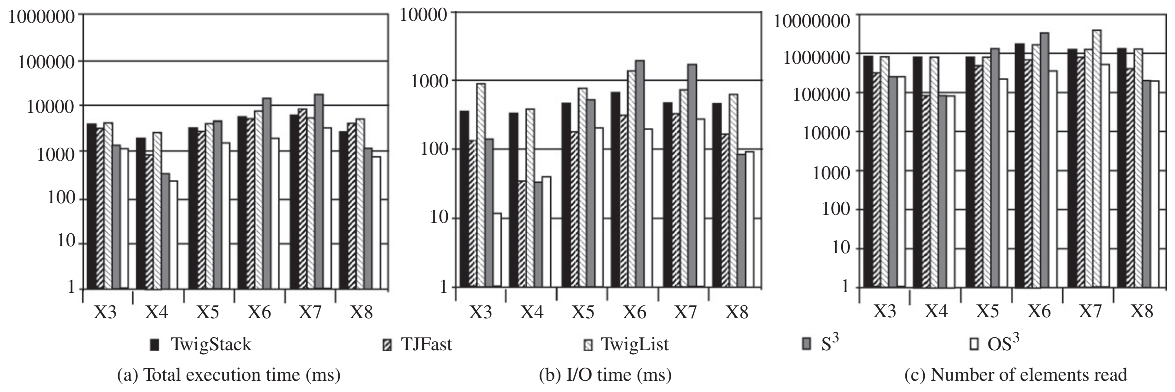

Fig. 15. Experimental results for XMark (scale 5).

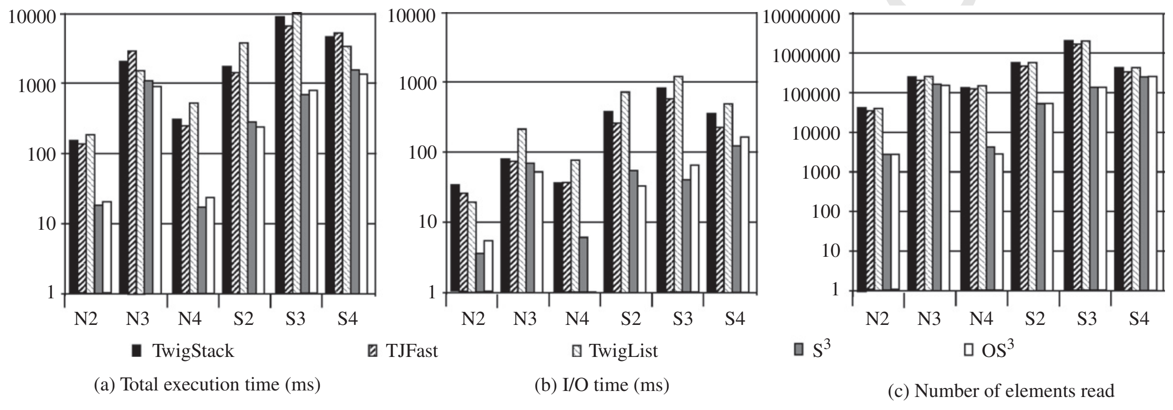

Fig. 16. Experimental results for Nasa and SwissProt.

5.4. Scalability analysis

We have also compared the scalability of our methods against *TwigStack*, *TJFast*, and *TwigList* in terms of document size and memory (maximum heap size of Java Virtual Machine). To analyze scalability concerning the document size, we created 18 XMark datasets with a scaling range from 0.1 to 9. Fig. 17a–c represents the scalability results for *X4*, *X6*, and *X7*, respectively. As can be seen, execution times for *TwigStack*, *TJFast*, and *TwigList* scale in a linear way w.r.t. dataset size, but they embody sub-linear behavior for S^3 and OS^3. As an exception, S^3 shows linear behavior on *X6*. As discussed above, S^3 has to fetch some elements 162 times, but still keeps the linear behavior.

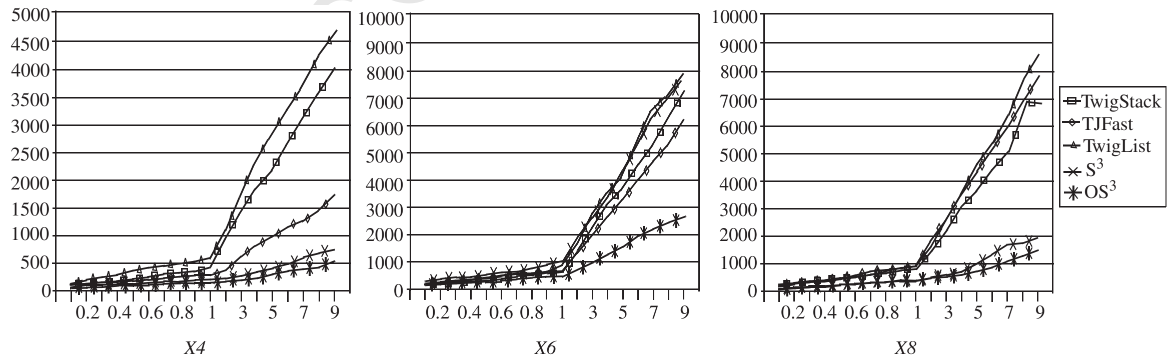

Fig. 17. Scalability of document size: total execution time (ms)

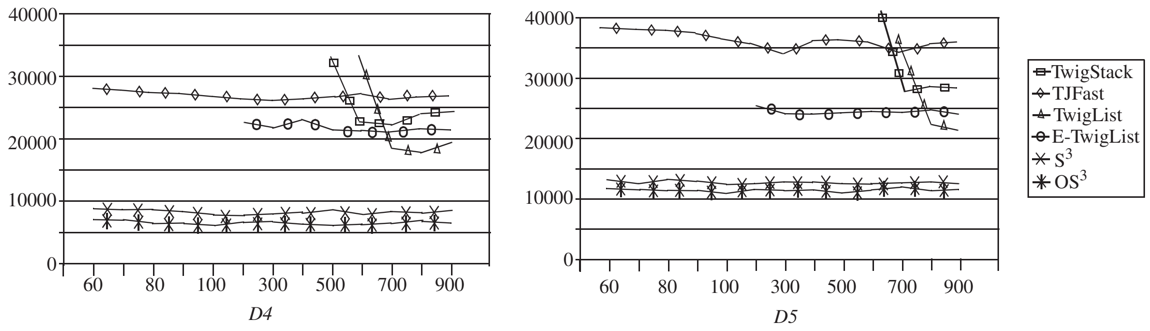

Fig. 18. Scalability of memory available: total execution time (ms)

We also did scalability tests in terms of memory available, where we performed our experiments for *D4* and *D5* (see Fig. 18) with 13 different heap sizes from 60 MB to 900 MB. Our results illustrate that *TwigList* needs substantial memory. For example, it cannot evaluate *D5* for heap sizes under 600 MB. We also implemented the external version of *TwigList*, called *E-TwigList*, which can only evaluate *D5* when the maximum heap size provided is more than 200 MB. It is worth noticing that *TwigStack* and *TJFast* should have same behavior in terms of memory size, because they produce the same amount of intermediate results. Because we enabled parallel execution of the matching and merging phases for *TJFast*, we achieved successful executions for it in all configurations. As depicted in Fig. 18, the execution time decreases when the maximum heap size is increased. The execution time for *TJFast* reaches its minimum and gets stable using a heap size of about 300 MB. The effect of increased memory size available is less for S^3 and OS^3; they reach their minimum execution time already for 100 MB. This observation shows that S^3 and OS^3 need less memory for query evaluation and behave better when the heap size is limited or when transactions need to share a fixed or limited amount of memory in a real multi-user environments.

6. Conclusions

In this paper, we reviewed some well-known QTP processing methods. *Structural Join* as the oldest method decomposes a QTP into its binary relationships and executes them separately. Its key drawback is the high amount of intermediate results produced during the matching process. *TwigStack* as a holistic method processes a QTP as a whole in two phases. During the first phase, *TwigStack* produces partial results for each QTP leg, whereas these partial solutions are merged in the subsequent phase to produce the final result. The main drawback of *TwigStack* is its expensive merging phase. *TJFast*, inspired by *TwigStack*, aims at improvements by reducing I/O. It uses an extension of the Dewey labeling method which enables the mapping of node labels to their related paths in the document. As a consequence, only potential target nodes of QTP leaves have to be fetched, but it is still burdened by the expensive merging phase. *Twig*^{2}*Stack* and its refined version *TwigList* evaluate QTPs without merging in a single phase, but they require more memory than *TwigStack* and *TJFast*. In the worst case, they have to load the entire document into the memory.

To overcome these problems, we proposed our method S^3. We emphasized the power of DeweyIDs and *QueryGuide* and showed that a *Structural Summary* enables for a QTP the derivation of an execution plan (SMP) to focus document (element index) access and reduce I/O cost as much as possible. The resulting SMP also facilitates the matching process. Enriched by information extracted from each MPs in the SMP, we can produce matches only by comparing DeweyIDs of elements related to QTP leaves without producing or accessing labels related to the inner QTP nodes. We can produce a full match containing labels related to all QTP nodes, just before the output of the final result. As a consequence, we produce only an insignificant amount of intermediate results which need less memory, too. Our scalability experiments limited by the available memory confirm that our method can also efficiently work in real multi-user XML databases. We also proposed an optimized version of S^3 (OS^3) to overcome situations where S^3 produces MPs which require repetitive access to some elements. Our scalability tests concerning document size revealed that OS^3 is the only method providing sub-linear behavior in our experiments, whereas S^3 behaves similarly, but exhibiting linear behavior in the worst case.

Acknowledgements

The first author wants to thank Professor Theo Härder (second author) for his kind scientific and non-scientific help during this work and the XTC project members – especially Christian Mathis, Karsten Schmidt, Sebastian Bächle, and Jose de Aguiar Moraes Filho – for their help during the implementation of the algorithms described in this paper. The authors are grateful for the hints of the anonymous referees, which improved the readability of this paper. The support of Andreas Bühmann while formatting the final version and that of Mayce Ibrahim Ali and Arthur Harutyunyan (International School of Graduate Studies) during the stay of the first author in Kaiserslautern is appreciated.

References

[1] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, Y. Wu, Structural joins: a primitive for efficient XML query pattern matching, in: Proceedings of the ICDE Conference, 2002, pp. 141–152.
[2] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, J. Simeon, XML Path Language (XPath) 2.0, W3C Working Draft, 2007. <http://www.w3.org/TR/xpath20>.
[3] T. Böhme, E. Rahm, Supporting efficient streaming and insertion of XML data in RDBMS, in: Proceedings of the Third International Workshop Data Integration over the Web, Riga, Latvia, 2004, pp. 70–81.
[4] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: Proceedings of the SIGMOD Conference, 2002, pp. 310–321.
[5] S. Chen, H.G. Li., J. Tatemura, W.P. Hsiung, D. Agrawal, K.S. Candan, Twig^{2}Stack: bottom-up processing of generalized tree-pattern queries over XML documents, in: Proceedings of the VLDB Conference, 2006, pp. 283–294.
[6] T. Chen, T.W. Ling, C.Y. Chan, Prefix path streaming: a new clustering method for optimal holistic XML twig pattern matching, in: Proceedings of the DEXA Conference, 2004, pp. 801–810.
[7] T. Chen, J. Lu, T. Ling, On boosting holism in XML twig pattern matching using structural indexing techniques, in: Proceedings of the SIGMOD Conference, 2005, pp. 455–466.
[8] S.-Y. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, C. Zaniolo, Efficient structural joins on indexed XML, in: Proceedings of the VLDB Conference, 2002, pp. 263–274.
[9] M. Dewey, Dewey decimal classification system. <http://www.mtsu.edu/vvesper/dewey.html>.
[10] M. Fontoura, V. Josifovski, E. Shekita, B. Yang, Optimizing cursor movement in holistic twig joins, in: Proceedings of the CIKM Conference, 2005, pp. 784–791.
[11] R. Goldman, J. Widom, DataGuides: enabling query formulation and optimization in semistructured databases, in: Proceedings of the VLDB Conference, 1997, pp. 436–445.
[12] T. Härder, M.P. Haustein, C. Mathis, M. Wagner, Node labeling schemes for dynamic XML documents reconsidered, Data and Knowledge Engineering 60 (1) (2007) 126–149.
[13] T. Härder, C. Mathis, K. Schmidt, Comparison of complete and elementless native storage of XML documents, in: Proceedings of the IDEAS Symposium, 2007, pp. 102–113.
[14] M.P. Haustein, T. Härder, An efficient infrastructure for native transactional XML processing, Data and Knowledge Engineering 61 (3) (2007) 500–523.
[15] H. Jiang, W. Wang, H. Lu, J. Xu Yu, Holistic twig joins on indexed XML documents, in: Proceedings of the VLDB Conference, 2003, pp. 273–284.
[16] H. Jiang, H. Lu, W. Wang, B.C. Ooi, XR-tree: indexing XML data for efficient structural joins, in: Proceedings of the ICDE Conference, 2003, pp. 253–264.
[17] M. Ley, DBLP Computer Science Bibliography. <http://dblp.uni-trier.de/xml/dblp.xml> (accessed 10.10.07).
[18] J. Lu, T. Chen, T.W. Ling, Efficient processing of XML twig patterns with parent child edges: a look-ahead approach, in: Proceedings of the CIKM Conference, 2006, pp. 533–542.
[19] J. Lu, T.W. Ling, C.Y. Chan, T. Chen, From region encoding to extended Dewey: on efficient processing of XML twig pattern matching, in: Proceedings of the VLDB Conference, 2005, pp. 193–204.
[20] Ch. Mathis, Extending a tuple-based XPath algebra to enhance evaluation flexibility, Informatik – Forschung und Entwicklung 21 (3) (2007) 147–164.
[21] P.E. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury, ORDPATHs: insert-friendly XML node labels, in: Proceedings of the SIGMOD Conference, 2004, pp. 903–908.
[22] L. Qin, J. Xu Yu, B. Ding, TwigList: make twig pattern matching fast, in: Proceedings of the DASFAA Conference, 2007, pp. 850–862.
[23] A.R. Schmidt, F. Waas, M.L. Kersten, M.J. Carey, I. Manolescu, R. Busse, XMark: a benchmark for XML data management, in: Proceedings of the VLDB Conference, 2002, pp. 974–985.
[24] University of Kaiserslautern: The XTC project. <http://wwwlgis.informatik.uni-kl.de/cms/index.php?id=36>.
[25] University of Washington: XML Repository. <http://www.cs.washington.edu/research/xmldatasets/>.

Sayyed Kamyar Izadi is a Ph.D. student at the Iran University of Science and Technology since 2003. He received his B.Sc. degree from the Isfahan University of Technology in 2001. He finished his M.Sc. degree at the Iran University of Science and Technology in 2003. He joined the DBIS research group lead by Prof. Härder and participated in the XTC project (a native XML database management system) for one year to the end of June 2008.

Theo Härder obtained his Ph.D. degree in Computer Science from the TU Darmstadt in 1975. In 1976, he spent a post-doctoral year at the IBM Research Lab in San Jose and joined the project System R. In 1978, he was associate professor for Computer Science at the TU Darmstadt. As a full professor, he is leading the research group DBIS at the TU Kaiserslautern since 1980. He is the recipient of the Konrad Zuse Medal (2001) and the Alwin Walther Medal (2004) and obtained the Honorary Doctoral Degree from the Computer Science Department of the University of Oldenburg in 2002. Theo Härder's research interests are in all areas of database and information systems – in particular, DBMS architecture, transaction systems, information integration, and Web information systems. He is author/coauthor of 7 textbooks and of more than 200 scientific contributions with >130 peer-reviewed conference papers and >60 journal publications. His professional services include numerous positions as chairman of the GI-Fachbereich "Databases and Information Systems", conference/program chairs and program committee member, editor-in-chief of Informatik – Forschung und Entwicklung (Springer), associate editor of Information Systems (Elsevier), World Wide Web (Kluver), and Transactions on Database Systems (ACM). He served as a DFG (German Research Foundation) expert and was chairman of the Center for Computed-based Engineering Systems at the University of Kaiserslautern, member of two joint collaborative DFG research projects DFG (SFB 124, SFB 501), and co-coordinator of the National DFG Research Program "Object Bases for Experts".

Mostafa S. Haghjoo is an associate professor at the Iran University of Science and Technology. He received his B.Sc. in mathematics and computer science from the Shiraz University in 1976. He received his M.Sc. degree in computer science from the George Washington University in 1978. He obtained his Ph.D. degree in computer science from the Australian National University in 1995.