S3: Processing tree-pattern XML queries with all logical operators

Sayyed Kamyar Izadi, Mostafa S. Haghjoo, Theo Härder

PII:	S0169-023X(11)00126-1
DOI:	doi: 10.1016/j.datak.2011.09.003
Reference:	DATAK 1351
To appear in:	*Data & Knowledge Engineering*
Received date:	24 May 2010
Revised date:	24 August 2011
Accepted date:	7 September 2011

S^3: Processing tree-pattern XML queries with all logical operators

Sayyed Kamyar Izadi
Department of Computer Engineering
Iran University of Science & Technology, Tehran, Iran
+98-21-73913309
izadi@iust.ac.ir

Mostafa S. Haghjoo
Department of Computer Engineering
Iran University of Science & Technology, Tehran, Iran
+98-21-73913309
haghjoom@iust.ac.ir

Theo Härder
Department of Computer Science
University of Kaiserslautern

D-67663 Kaiserslautern, Germany
+49-631-205-4030
haerder@informatik.uni-kl.de

ABSTRACT

XML is a tree-based data representation format which combines data and structure. Therefore, XML queries not only contain predicates to filter data but also refer to relationships between document elements searched. The existing elements in an XML query are connected to each other using a tree-pattern structure, called Query Tree Pattern (QTP). Finding elements of a document, which satisfy the given QTP, is the main task during query execution. To optimize this processing, we presented two methods in [13]. Instead of directly executing the QTP against the document, our methods first evaluate a guidance structure, called *QueryGuide*. Using the extracted information, called *match pattern*, we provided a focused document access and minimized the required I/O. However, we only supported the logical operator *AND* (called *AND*-QTPs).

In this paper, we use a new structure, called *Evaluation Tree*, to execute QTPs. We also extend our method to support QTPs having logical operators *OR*, *XOR*, and *NOT*. Parsing QTPs into some AND-QTPs is typically assumed non-efficient. To process QTPs having logical operators *OR* and *NOT*, we therefore parse them but we use an efficient method to prevent redundant I/O and QTP matching. This is done by optimizing the selection of match patterns which were derived from the *QueryGuide* during QTP parsing. As a result, QTP execution is not inefficient anymore.

1. INTRODUCTION

The flexibility of XML provides suitable data representation formats for many applications, especially for those dealing with semi-structured data. Furthermore, the emergence of XML database management systems (XDBMSs) is a response to the growing popularity of XML in various domains, where collections of huge XML documents have to be managed. Querying such data volumes poses new challenges, especially related to the structure of XML documents, which combine tree structure with document content. XPath [2] and XQuery [3], the two most popular query languages in the XML domain, reflect the XML tree structure in their syntax using path expressions. As a consequence, XML queries enable the specification of so-called query tree patterns (QTP). Such QTP evaluations on large XML documents, however, are very expensive, because all XML fragments matching a given QTP have to be located.

Consider the query Q_1: *//A[.//B]/C//D*. Figure 1 represents the QTP of Q_1, which has two branches: *A//B* and *A/C//D*. To evaluate Q_1, it is sufficient to access the document nodes related to elements occurring in Q_1. These nodes may be part of the final result if they satisfy the related path conditions and also have counterparts to satisfy the other branches specified in the QTP. In the Q_1 example, document nodes related to *B* must have an *A* element as their ancestor to satisfy the left branch. Furthermore, such *A* elements must have a *C* element as child and, in turn, at least a *D* element as descendant of this *C* element. QTP evaluation may become even more complicated and expensive if more than two branches occur in a query. Further, branches of a QTP may be connected by logical operators other than *AND*. In particular, branches of QTPs may contain the logical operator *NOT*.

1.1 Related Work

We have addressed the problem of QTPs having branches connected only by the logical operator *AND* (*AND*-QTPs) in [13]. Some methods have been proposed to efficiently evaluate *AND*-QTPs against huge XML documents. *Structural Join* is one of the first methods [1] decomposing a given QTP into its binary relationships, where each binary relationship, e.g., *C//D*, is separately executed against the document thereby producing huge volumes of intermediate results. Eventually, the final result is formed by combining these intermediate results. Some other methods such as those described in [9][15] attempt to improve the efficiency of the *Structural Join*. *TwigStack* [4] provided a novel solution avoiding the decomposition of a query into its basic relationships. Instead, intermediate results are evaluated for each QTP leg (root-to-leaf path) in the first phase. These single-path matches are then merged to produce the final result in the second phase. To improve *TwigStack*, various index structures were proposed [7] [8] [16]. Furthermore, *TwigOptimal* [11] introduced the idea of jumping over non-qualified elements in the indexes to achieve superior performance.

Inspired by *TwigStack*, *TJFast* [20] reduced I/O using a refined version of the Dewey labeling method (see Section 1.2). Each Dewey label enables the identification of the entire path from the root to its related node in the document. Therefore, *TJFast* could easily produce partial results related to each QTP leg by only accessing the potential target nodes of QTP leaves.

The main problem with these methods is their expensive merging phase. As an answer to this drawback, *Twig*^{2}*Stack* [6] and its

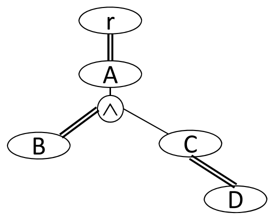


Figure 1. Related QTP of query Q_1.

refined version *TwigList* [22] eliminate the merging phase. But, in turn, these methods introduce a severe shortcoming: they have to fetch the entire document into main memory in the worst case.

All the above mentioned algorithms are limited to processing AND-QTPs. The importance of logical operators such as *OR* and *NOT*, therefore, attracted researchers to design more powerful solutions. *GTwigMerge* [17] handles QTPs containing logical OR operators. QTP processing is performed by converting *AND*-QTPs using the concept of *OR-Blocks*. Similar to *TwigStack*, *GTwigMerge* is a two-phase method, where potential target nodes of all QTP nodes have to be accessed in the first phase. *PathStack¬* [18] extends the *PathStack* [4] method, which only evaluates single path queries containing logical *NOT* operators. *TwigStackList¬* [26] is an extension of *TwigStack*, which enables processing of QTPs containing logical *NOT* operators. In addition to chained stacks used in *TwigStack*, this method accelerates QTP processing by a look-ahead approach, where potential target nodes are fetched for each QTP node. Finally, *AllTwigMerge* [5] is a method, inspired by *GTwigMerge*, designed to evaluate QTPs containing logical *OR* and *NOT* operators by converting QTPs into a normalized form.

In this paper, we use the following concepts whose detailed definitions can be found in [13]:

- XTS: is a logical representation of an XML document.
- DeweyID: is a kind of label assigned to XTS nodes. Label of each node contains label of its parent.
- *Structural Summary*: is a structure summarizing existing paths of an XML document.
- *QueryGuide*: is a combination of a *Structural Summary* in addition to a *Reference Section* which indexes document nodes related to each node of the *Structural Summary*. Figure 2 represents an XTS representation of an XML document and its related *QueryGuide* is shown in Figure 3.
- Match Pattern: is a structure enabling focused document access. Execution of a QTP against the document's *Structural Summary* results in a set of match patterns, called SMP.

1.2 Contribution of this paper

One of the major ideas to improve QTP performance is to reduce I/O. We prevent accessing document nodes that definitely do not participate in the final result or if their related information can be evaluated from other nodes accessed. To reach this goal, we proposed the *QueryGuide* in [13] which contains a summary of the document structure. Then, the execution of QTPs against the *QueryGuide* enables us to identify which parts of the document are mandatory for the evaluation of the final matches.

Furthermore, we use the Dewey labeling method [12] to assign a label to each document node. Therefore, each node accessed provides information related to all its ancestors and helps to eliminate I/O previously needed to access their information.

In [13], we proposed two methods S^3 and OS^3 for processing *AND*-QTPs. In this paper, we refer to them as S^3.v0 and S^3.v1, respectively; we improve these methods and enhance them by the logical operators *OR*, *XOR*, and *NOT*. It is worth noting that *GTwigMerge*, *PathStack¬*, *TwigStackList¬*, and *AllTwigMerge* are based on the *TwigStack* method. As we reported in [13], our methods S^3.v0 and especially S^3.v1 clearly outperform *TwigStack* for *AND*-QTPs. In this paper, our main contributions and improvements are:

- *Evaluation Tree* as a new structure which is formed based on the given QTP[1] and is fed by document nodes to evaluate final matches.
- Execution of QTPs with logical operators *OR* and *NOT* by parsing QTPs into AND-QTPs using the *QueryGuide* to achieve an efficient evaluation.
- Introduction of "super-patterns" to decrease the number of match patterns necessary to execute QTPs having logical operators *OR* and *NOT*.
- Grouping of super-patterns to reduce redundant I/O.

The rest of this paper is organized as follows: Basic concepts used in this paper are described in Section 2. A brief description of methods proposed in [13] is reviewed in Section 3. In Section 4, we introduce our new structure, called *Evaluation Tree*, and its use for QTPs having logical *OR* operators. Section 5 describes how we support the logical operator *NOT*. In Section 6, we present our method for the logical operators AND, OR, and NOT together. Finally, we summarize the experimental results in Section 7 and conclude our work in Section 8.

2. PRELIMINARIES

In order to represent an overview of our previous methods and make this paper more self-contained, it is necessary to take a quick look at some definitions and concepts. Details can be found in [12][13].

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 ($V(r)$="root").

Definition 2. A QTP 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: $Q=\{x \mid \exists n \in N_X$, T(n)="element" $\vee$ $T(n)$="attribute", $V(n)=V''(x)\}$
- $O=\{\wedge, \vee, \neg, \oplus\}$ 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.

[1] For sake of simplicity, we focus on QTPs with nodes having children connected by *only one* of the *AND* or *OR* logical operators. Covering QTPs with nodes having combinations of logical operators does not have a major impact on our algorithms (see Appendix B).

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>
```

b

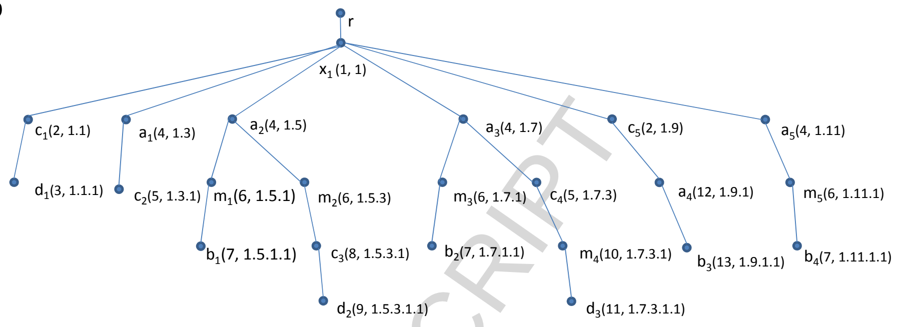


Figure 2. (a) A simple XML document (b) XTS representation of the document (X_1) labeled by DeweyIDs

- $E'' \subset (Q \cup O)\times(Q \cup O)$ represents branches of QTP. All leaves of the QTP are *query* nodes.
- *U:* $Q\times\{$"A-D", "P-C"$\}$ indicates a kind of relationship between a query node *q* and its nearest query node among the 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 $\rightarrow$ *String* returns the value of a node.
- *C:* $Q\times N_X \rightarrow \{true, false\}$ is a Boolean function deciding whether or not a node $n \in N_X$ satisfies the constraints associated with 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, <):

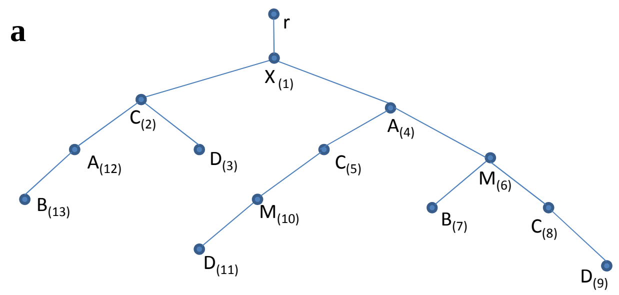
a) $PTN(q)=\{n \mid n \in N_X, V(n)=V''(q) \wedge C(q, n)\}$

b) $\forall\, n_1, n_2 \in N_X$: $n_1 < n_2$ iff n_1 is visited earlier than n_2 in a pre-order traversal through *X*.

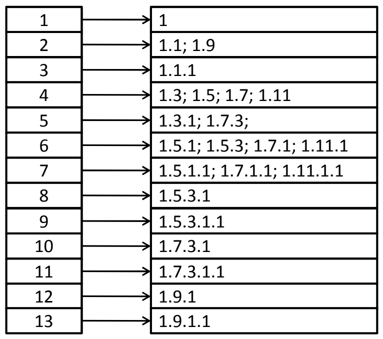
DeweyID: Performing query evaluation against a huge XML document with acceptable performance needs the document to be stored in a format that provides facilities like indexes [14]. The first step is to assign a label to each node in the document. In this way, XML documents are stored in a structured format. Comparisons of labeling schemes and their empirical evaluation [12] led us to use a prefix-based scheme, based on the concept of Dewey order [21], for the labeling of tree nodes. Historically, Dewey labeling was first used in libraries providing a better way to find items on the shelves [10]. Dewey labels used in the XML database domain consist of so-called *divisions* (separated by dots) representing the node path from the document root to the node itself. The label of each node is constructed by adding a new division to the label of its parent; therefore, it contains the labels of all its ancestors. Dewey labels also contain other valuable information. The odd-numbered divisions in a label enable the derivation of the depth of a document node. Furthermore, the relation between two nodes such as ancestor-descendant, parent-child, or their order in the document can easily be identified just by comparing the labels of the respective nodes. Several labeling methods based on Dewey labeling were proposed. Our mechanism, referred to as DeweyID, is characterized by some distinguished features described in [12]. DeweyIDs have nice and practical properties. For example, they are immutable, that is, they allow the assignment of new IDs without reorganizing the IDs of present nodes. Figure 2 represents a sample XML document labeled by DeweyIDs.

QueryGuide: Evaluation of queries against huge XML documents may require lots of document scans. In such situations, indexes in the form of B*-trees can be a solution to minimize the volume of I/O. However, the particular form of XML documents implies that there is a need to capture the structure of XML

a



b



1	1
2	1.1; 1.9
3	1.1.1
4	1.3; 1.5; 1.7; 1.11
5	1.3.1; 1.7.3;
6	1.5.1; 1.5.3; 1.7.1; 1.11.1
7	1.5.1.1; 1.7.1.1; 1.11.1.1
8	1.5.3.1
9	1.5.3.1.1
10	1.7.3.1
11	1.7.3.1.1
12	1.9.1
13	1.9.1.1

Figure 3. QueryGuide for X_1 in Figure 2: (a) *Structural Summary* S_1(b) reference section

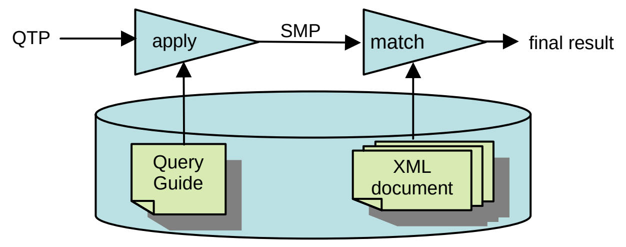


Figure 4. Overview of the S^3 methods.

documents beside their content. *QueryGuide* explained in [13] is our solution. It consists of two parts: *Structural Summary* and *Reference Section*. *A Structural Summary* is itself an XTS object representing the XML structure of a single document. Each path instance in a given document is mapped to a single path in the *Structural Summary*, called its path class. Therefore, a *Structural Summary* covers the entire path instances of the document and, in turn, each node in the *Structural Summary* is the representative of all *instance nodes*, which are the nodes in the document having the same path labeled by a unique CID (*Class ID*). The *Reference Section* indexes DeweyIDs of instance nodes related to *Structural Summary* nodes (CIDs).

Lemma 1[13]**.** 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.

3. OVERVIEW OF S^3.v0 AND S^3.v1[2]

The power of our methods is founded upon the previously introduced concepts, *DeweyIDs* and *QueryGuide*, which provide valuable information for QTP processing. The execution of a QTP against the related *Structural Summary* of the *QueryGuide* leads to a set of *Match Patterns* (MPs). This set (referred to as SMP hereafter) enables focused document access. Using the strength of Dewey labels, the extracted nodes are efficiently compared and joined in the matching process based on MPs. Figure 4 shows the overall process of QTP matching in our method.

3.1 Matching Process

The volume of I/O during a matching process is a critical issue for the efficiency of a QTP processing method. We have optimized it by the use of DeweyIDs and SMPs. In our method, only the extraction of potential target nodes of QTP leaves is essential. The DeweyID of an extracted node can be used to produce the label of potential target nodes of inner QTP nodes whenever required. Furthermore, the execution of a QTP against the *Structural Summary* produces an SMP. For each QTP leaf, the SMP contains a useful subset of CIDs of potential target nodes needed to be extracted. The corresponding nodes of these CIDs satisfy one of the QTP legs in the document and often enable the finding of other nodes to produce a match for the given QTP as well.

Example 1. Consider QTP *Q1* (Figure 1) and XTS object X_1 (Figure 2). *Q1* is first executed against *Structural Summary* S_1 (Figure 3(a)). The result is a single match (A_4, B_7, C_5, D_{11}). As a result, only those members of PTN(*B*) having CID 7 and members of PTN(*D*) having CID 11 have to be extracted. A closer look at X_1 indicates that d_1 with CID 7 has no *A* element as an ancestor to satisfy *D*-leg (*//A/C/D*) of *Q1*, and d_2 cannot satisfy the *D*-leg because c_3 is not a child of a_2. On the other hand, S_1 reveals that, while all members of PTN(*B*) can satisfy the *B*-leg, *B* elements with CID 13 have no counterpart to match *Q1*. This fact is observable in X_1. ♣

As illustrated in the above example, the execution of QTPs against a *Structural Summary* provides valuable information to optimize the evaluation of QTPs. Based on the *Structural Summary* definition, we can claim that the use of SMPs, as input of the matching process, does not discard any potential final match as proved in the following theorem.

Theorem 1[13]**.** *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.* ♣

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. Each MP has enough information to produce all final matches belonging to it. Therefore, QTP is first executed against the document's *Structural Summary* in S^3.v0. Considering a resulted match pattern *MP*, for each leaf l_i, a stream of sorted DeweyIDs, referenced in the *Reference Section* by CID of *MP*(l_i), is then extracted. The matching process related to *MP* starts by joining the extracted nodes of the first two leaves. Consider two leaves l_i and l_j and the pair of DeweyIDs (d_i, d_j). d_i satisfies the l_i-leg, because MP(l_i) satisfies the l_i-leg (Lemma 1). The same is true for d_j. d_i and d_j can be joined if they have the same ancestor related to the nearest common ancestor (NCA) of l_i and l_j (*jp*) in the QTP. This is possible if d_i and d_j have the same prefix w.r.t. the level of *jp* in the *Structural Summary*. The matching process (related to *MP)* continues by joining the produced intermediate results with the nodes related to the next leaf. The matching process is finished when all leaves of the QTP are covered.

Example 2. Consider *Structural Summary* S_2 and QTP Q_2 in Figure 5. We first execute Q_2 against S_2 which results in a single match pattern *MP*(E_2, N_3, G_4, H_5, L_8). Q_2 has three leaves *G*, *H*, and *L*. Hence, in the matching process three streams of potential target nodes w.r.t. G_4, H_5, and L_8 are 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}*. As explained above, the first two leaves of Q_2 are selected (*G* and *H*). All streams related to these leaves are joined. *N* is the NCA of *G* and *H*. Therefore, those nodes could be joined which have the same *N* node as their common ancestor. The entire nodes related to G_4, H_5 (*RF(4)* and *RF(5)*) have an *N* node as their ancestor because N_3 is an ancestor for both G_4 and H_5. N_3 is placed in the third level of S_2. This means that the prefixes with length 3 of DeweyIDs in *RF(4)* and *RF(5)* is the DeweyID of an *N* node with CID 3. As a consequence, those nodes of *RF(4)* and *RF(5)* can be joined that have the same prefix with length 3. The resulting pairs have to be joined with nodes related to the third leaf of Q_2 (*RF(8)*). H_5 and L_8 have E_2 as their common ancestor, which is placed in the second level of S_2. As a result, we can join an *l* node with a pair of (*g*, *h*) if the prefix (with length 2) of the DeweyID of the *l* node and the *g* node are the same. This prefix is the DeweyID of an *e* node related to E_2 as common ancestor of *l*

[2] In this paper, we refer to S^3 and OS^3, our proposed methods in [13], as S^3.v0 and S^3.v1, respectively.

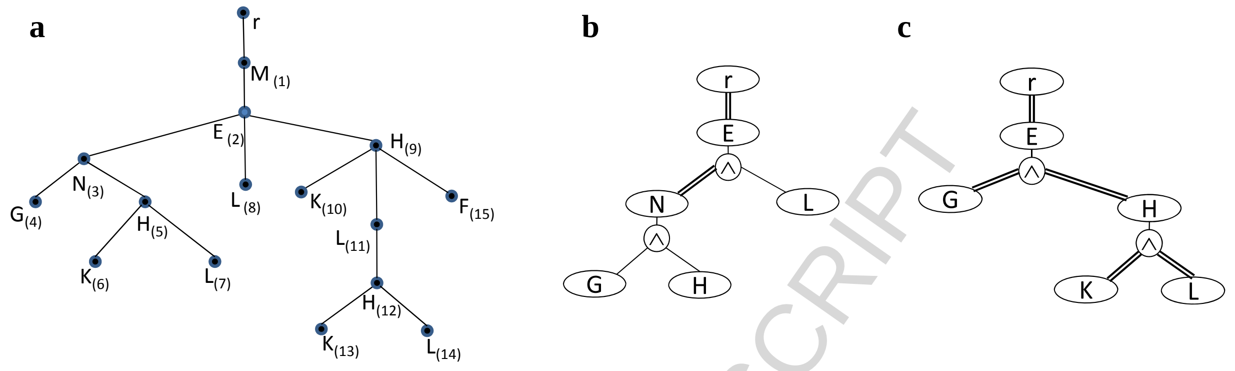


Figure 5. (a) A sample *Structural Summary* (S_2); (b) QTP Q_2; (c) QTP Q_3;

and g. It is straightforward to show that *e* is also the ancestor of g. The relevant matching process is depicted in Figure 6.♣

Note that matching is done by a pipelining strategy as illustrated in Figure 6. Join results are sent to the next joining step to form a more complete match immediately after they are produced. These intermediate results are discarded as soon as they are used in the matching process. Therefore, the entire matching process is carried out without consuming a huge amount of main memory.

3.2 I/O Optimization

S^3.v0 executes QTPs against the *Structural Summary* of the *QueryGuide* to perform a focused search to minimize the volume of I/O during the matching process. However, S^3.v0 misses this goal, if a common leaf node is repeated in a large number of MPs of the resulting SMP. To minimize the repeated I/O accesses, we proposed an enhanced version of S^3, named S^3.v1. In this method, MPs having common nodes in an SMP are found and arranged to form so-called grouped MPs (GMP). Each GMP is responsible to produce the entire results related to its wrapped MPs. In order to form GMPs, the distinct number of node occurrences in the SMP related to each QTP leaf is counted. The leaf, whose related nodes have minimum distinct occurrence, has maximum repeated nodes in the SMP and is selected to group MPs. Those MPs having a common node related to the selected leaf are grouped into a separate GMP. Inspired by S^3.v0, for each GMP, a matching process is run, in which, for each leaf l_i, a stream of nodes related to the GMP(l_i) is assigned. However, in S^3.v1, GMP(l_i) corresponds to a set of *Structural Summary* nodes - not a single node. Furthermore, based on each wrapped MP in GMP, there is probably a different join-point level. To cover these problems, for each GMP(l_i), a sorted stream of nodes is created. For each CID in GMP(l_i), a stream of nodes is created and, in each access, the minimum node among these streams is selected and returned. The next difficulty is the existence of different join-point levels for each leaf pair. As a solution, the minimum join-point level among different wrapped MPs is selected for each leaf pair. Now the matching process is done in almost the same way as in S^3.v0.

Contrary to S^3.v0, the joining process in S^3.v1 may produce false-positive outputs. This problem arises when the join point of a pair of nodes related to a wrapped MP is searched in higher than the actual level of the *Structural Summary*. Moreover, the node stream used in S^3.v1 contains nodes related to different MPs. Consequently, a pair of nodes not related to the same MP is probably considered as output (false-positive result). This fact indicates that there is a need to check the output of the joining process in S^3.v1 to match it against wrapped MPs. If none of the wrapped MPs matches an output result, then this output is

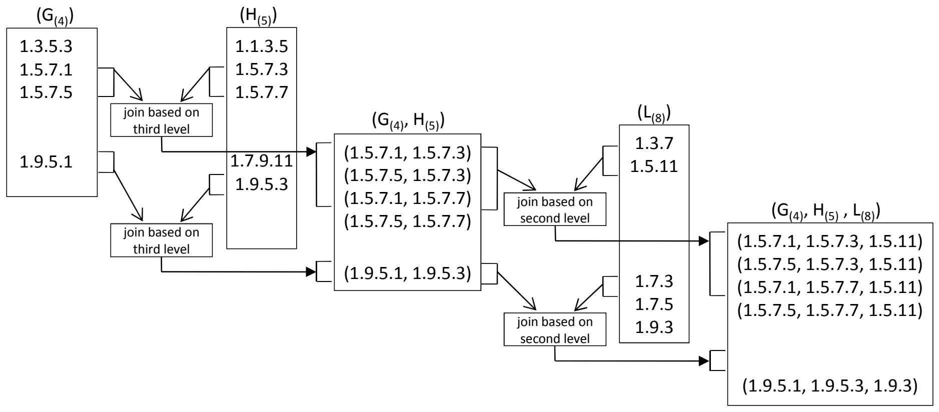


Figure 6. Matching process for Q_2.

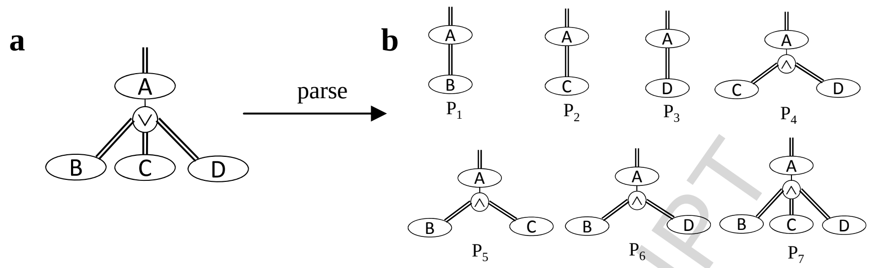


Figure 7. (a) QTP Q_4; (b) its parsed AND-QTPs.

discarded. On the other hand, output of the joining process in S^3.v1 may match more than one wrapped MP. This is possible when two or more wrapped MPs have the same node corresponding to some QTP leaves. In this situation, the nodes related to the inner QTP nodes are different and, for each wrapped MP matched, a separate result is formed.

Example 3. Consider QTP Q_3 and *Structural Summary* S_2 in Figure 5. Execution of Q_3 against S_2 results in the following SMP: {$mp_1(E_2, G_4, H_5, K_6, L_7)$, $mp_2(E_2, G_4, H_9, K_{10}, L_{11})$, $mp_3(E_2, G_4, H_{12}, K_{13}, L_{14})$, $mp_4(E_2, G_4, H_9, K_{10}, L_{14})$, $mp_5(E_2, G_4, H_9, K_{13}, L_{11})$, $mp_6(E_2, G_4, H_9, K_{13}, L_{14})$}. A quick look over the resulting SMP indicates that G_4 is repeated in the six match patterns and, therefore, S^3.v0 repeatedly extracts the target nodes of G_4 and the matching process considers them six times. In S^3.v1, since nodes related to leaf *G* have minimum distinct occurrence (only G_4), the resulting SMP is transformed to a grouped SMP (SGMP) based on leaf *G*. 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 nodes related to G_4 (nodes with CID 4) are fetched only once in S^3.v1 instead of six times done by S^3.v0.

The join-point 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, the joining process in S^3.v1 returns the 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 can't 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 is m_2 which can't match any MPs. The problem with m_2 is that, although 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 in the joining process. The last output is m_3, which matches both mp_3 and mp_6. ♣

4. PROCESSING QTPs WITH *OR* AND *XOR* NODES

In this section, we demonstrate how to process QTPs having *OR* nodes. The most straightforward method to process such QTPs is to parse them to form some *AND*-QTPs. The final result is formed by merging execution results of each parsed *AND*-QTP.

Definition 4. Consider a given QTP *Q*. The *output nodes* of *Q* are a subset of query nodes of *Q*, called *Output(Q)*, such that each node does not have any operator other than logical *AND* in its ancestors. Furthermore, the final results matching *Q* only have DeweyIDs w.r.t. members of *Output(Q)*. ♣

The methods S^3.v0 and S^3.v1 are specially designed for *AND*-QTPs. It is possible to use these methods for processing QTPs containing *OR* nodes by parsing the QTP into some *AND*-QTPs. We refer to this type of QTP processing as S^3.v2.

Parsing a QTP into *AND*-QTPs is not an optimal method for processing QTPs containing *OR* nodes. While only potential target nodes of QTP leaves are accessed during the matching process, these nodes may be accessed w.r.t. most of resulting *AND*-QTPs. This imposes a lot of I/Os and matching processes. The result of each *AND*-QTP may not be disjoint. Consequently, lots of those operations during QTP evaluation are redundant.

Example 4. Consider QTP Q_4 in Figure 7a and XML document X_1 (Figure 2) and its *Structural Summary* in Figure 3(a). Parsing Q_4 yields seven *AND*-QTPs shown in Figure 7b. Nodes with CID 7 are accessed nine times because of the parsed QTPs P_1, P_5, P_6, and P_7 (one time for P_1, two times for P_5 and P_6 and four times for P_7). These redundant document accesses do not necessarily produce useful results. For example, (a_2, b_1) and (a_3, b_2) are results of P_1 and (a_2, b_1, c_3) and (a_3, b_2, c_4) are results of P_5. We have to discard results of P_1 because the above results of P_5 cover them. Moreover, the above results of P_5 will be discarded by results of P_7. However, (a_5, b_4) is one of the P_1 results, which does not have any counterpart in P_5. This shows the contradiction between the necessity of executing each parsed QTPs against the *Structural Summary* to identify all possible results and the operational redundancy arising because of this need.♣

An idea to solve this redundancy is to use an approach inspired by one used in S^3.v1 [13]. By wrapping MPs into GMPs, we can avoid most of the redundant I/Os. However, the matching process used in S^3.v1 is not easily applicable here. In S^3.v2, parsed QTPs are executed separately and then their results are checked to see if they are already covered by results coming from other parsed QTPs. Producing GMPs based on the entirety of resulting MPs of parsed QTPs is applicable, but the main problem is how to process these GMPs. The wrapped GMPs in this situation, contrary to GMPs in S^3.v1, may be related to different QTPs with different leaves.

4.1 Evaluation Tree

The matching process in S^3.v1 is based on wrapped MPs corresponding to a single *AND*-QTP. The process starts by joining nodes related to the first two QTP leaves and continues to cover the entire leaves. In S^3.v2, the related QTP of each MP is known and the matching process is formed based on the related QTP leaves. If we form GMPs, then each GMP probably contains MPs, which are related to some QTPs having different leaves. To form the matching structure, we have to build a QTP covering all related QTPs of a GMP. The matching process in S^3.v1 returns results when joinable nodes are present for each leaf. But when QTP leaves are connected by a logical *OR*, there is no need for all leaves to have a joinable node. Consequently, considerable changes are needed in the matching process of S^3.v1 to overcome its defects when processing *OR* nodes. Therefore, a new join approach may be the better solution. For this purpose, we form a processing structure called *Evaluation Tree*.

Definition 5. For a given QTP *Q*, its related *Evaluation Tree ET* is a summarization of *Q*. The root and each inner node of *ET* are associated to join points of *Q*, and its leaves are associated to leaves of *Q*. The ancestor-descendant and parent-child relationships between *Evaluation Tree* nodes are the same as the relationships of their associated QTP nodes in *Q*. However, they may be reduced to a parent-child relationship. ♣

It is worth noting that we refer to the QTP node associated to an *Evaluation Tree* node *E* as QN(*E*) and upper nodes of QN(*E*) up to the first upper join point or root of the QTP as UQN(*E*). We use the *Evaluation Tree* to perform the matching process related to a match pattern. This tree is formed based on the QTP structure shown in Figure 21 using pseudo-code (function *getEvalTree*). The matching process begins with the extraction of target nodes of QTP leaves by their corresponding *Evaluation Tree* leaves. Then, the inner nodes are used to join the results of their children, which may be leaves of the tree or other inner nodes. When a leaf retrieves a target node (i.e., its label) of the associated QTP leaf, it can use this label and related MPs of the *Evaluation Tree* to produce labels related to the associated QTP node of its parent and QTP nodes in between. Therefore, when an inner node of *Evaluation Tree IE* reads results of its children, they contain the label related to QN(*IE*). As a result, considering the kind of logical operator associated to the QN(*IE*), *IE* has enough information to join its inputs and enrich join results by labels related to UQN(*IE*).

Example 5. Again consider QTP Q_2 in Figure 5(b) and the sample target nodes presented in Example 2. The corresponding *Evaluation Tree* of Q_2 is depicted in Figure 8. Obviously, the resulting *Evaluation Tree* has a similar structure as Q_2, because all nodes of Q_2 are join points or leaves. Node *G* reads the target nodes of G_4. Because UQN(*G*) = {*N*} and N_3 is placed in the third level of S_2, node *G* enriches the target nodes read with their prefix having length 3 and sends them to node *N*. As can be seen, node *G* reads 1.3.5.3 and then combines it with its prefix of length 3 as DeweyID related to node *N* (1.3.5, 1.3.5.3). The same procedure is done in Node *H*. Therefore, sub-results read by Node *N* have labels related to *N* and any combination of sub-results having the same label related to QTP node *N* are joined with each other. For example, node *G* has two output elements having DeweyID 1.5.7 related to node *N*. Furthermore, node *H* has two output elements having DeweyID 1.5.7. Since these sub-results have the same DeweyID related to node *N*, they are joined in node *N* of the *Evaluation Tree* producing four sub-results having DeweyIDs related to nodes *G*, *H*, and *N*. Again sub-results produced in node N have to be enriched. Since UQN(*N*) = {*E*} and E_2 is placed in the second level of S_2, the join results in node *N* are enriched with the prefix having length 2 of the labels related to QTP node *N*.

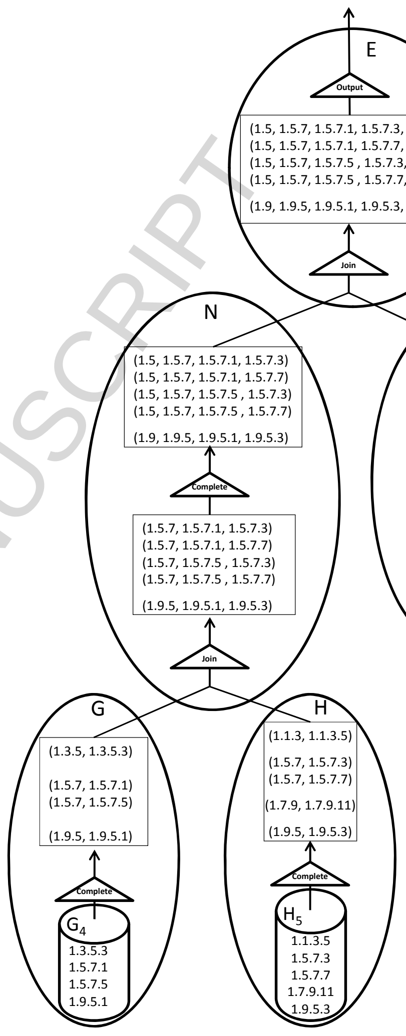


Figure 8. *Evaluation Tree* used to process Q_2.

Finally, node *E* reads the sub-results from node *N* and *L* to produce the final results. ♣

As depicted in the above example, the leaves of the *Evaluation Tree* are responsible to extract target nodes of their associated QTP leaf. Each inner node is responsible to join its inputs and send results to its parent. The final results are produced by the root of the tree. In other words, using a pipeline strategy, the root of the tree gets the sub-results from its children and joins them to produce the final results. Children of the root follow the same procedure to produce their related results, as well. This process spreads recursively from the root to the leaves of the *Evaluation Tree*. We refer to S^3.v3 as a method using this type of joining process.

4.2 Construction of an Evaluation Tree

In order to process a given QTP *Q* in S^3.v3, an *Evaluation Tree* is first formed based on *Q* and then *Q* is parsed to produce its related *AND*-QTPs (similar to S^3.v2). The match patterns, which are the results of executing the parsed QTPs against the document's *Structural Summary*, are used to feed the *Evaluation Tree*. For each leaf *l* of the *Evaluation Tree* and a given match pattern *MP*, a target node stream corresponding to *MP*(*l*) is considered. If *MP* does not have a member corresponding to *l*, an empty input is assigned to *l*. Since these kinds of leaves are those leaves of *Q* having at least one logical *OR* in their path to the root of the *Q*, null inputs do not have any side effect on the results related to these types of match patterns. Therefore, there is no need to create specific *Evaluation Trees* related to each parsed QTP. *Evaluation Trees* will have the same structure, but their input streams will be set based on the related match pattern. This kind of joining process opens the way to deal with logical operators other than *AND*. Each node in an *Evaluation Tree* retrieves output results of its children simultaneously. Therefore, the decision whether a set of input results would be joined or not can be simply made based on the kind of relationship between a node and its children.

When a QTP is given, it is traversed from the root to the first join-point node (lines 43-49 in Figure 21). Traversed nodes (upperNodes in Figure 21), which have only one child (forming a direct path), are assigned to the *Evaluation Tree* node corresponding to the join point. Therefore, each node *E* in the *Evaluation Tree* has enough information to produce proper DeweyIDs for UQN(*E*). Each *Evaluation Tree* leaf *LE* retrieves its inputs from its assigned node stream and, then, adds proper DeweyIDs for UQN(*LE*) in the QTP (lines 6-17 in Figure 23). As a result, each inner node of an *Evaluation Tree IE* receives sub-matches containing a DeweyID related to QN(*IE*) from its children. Based on the specified logical operator, the sub-results are joined and then the proper DeweyIDs for each QTP node of UQN(*IE*) are produced and inserted into the join results. These results are sent to the parent of the *Evaluation Tree* node. Consequently, the root of an *Evaluation Tree* produces the final result that matches the entire main QTP.

4.3 Matching Process using an Evaluation Tree

Figure 23 depicts how logical operators *AND* and *OR* are processed. If children of an inner node *IE* of an *Evaluation Tree* are connected to it by *AND* operators, joining sub-results of children is possible when all of them contain the same DeweyID related to QN(*IE*) (procedure processAND in Figure 23). As a result, sub-results of children have to be checked two-by-two. Consider children of *IE* as C_1, ..., C_n, if the DeweyIDs of two children C_i and C_{i+1} related to QN(*IE*) (d_i and d_{i+1}) are equal, then d_{i+1} and d_{i+2} are compared. If d_i is greater than d_{i+1}, the output of C_{i+1} is fetched and skipped to reach a sub-match that makes d_{i+1} equal or greater than d_i (procedure skip in Figure 22). When C_i and C_{i+1} are compared, if d_i is smaller than d_{i+1}, it means that all d_j, $j \leq i$, are smaller than d_{i+1}. Consequently, the output of each C_j is skipped (using procedure *skip* in Figure 22) and the process starts from the first child (C_1). We refer to this process as balancing of children for processing logical *AND* (see procedure *ANDBalance* in Figure 23). The balancing procedure continues until all C_is have the same d_i and get balanced or the output of one child is finished.

If C_is get balanced, then we can join d_is, but each C_i may have other sub-matches having the same DeweyIDs related to QN(*IE*) and equal to d_i. These sub-matches can also be used to produce more than one join result. As a consequence, when children get balanced, a list of sub-matches having the above feature is formed for each of them (procedure *advance* in Figure 22). All members of these lists are joinable, because they have the same DeweyID related to QN(*IE*). These members are joined using a Cartesian product (procedure *xProduct* in Figure 23), preparing outputs of *IE* which will be sent to its parent. The outputs will be enriched by DeweyIDs related to each node in UQN(*IE*) based on the DeweyID related to QN(*IE*), before sending an output to the parent of *IE* (procedure *completeToUpperJoinPoint* in Figure 22).

The main difference between processing *OR* and *AND* operations is related to their balancing process. An inner *Evaluation Tree* node *IE* carrying a logical *OR* joins sub-matches of its children as its outputs even when they all do not have the same DeweyID related to QN(*IE*). As a result, the balancing process will be finished as soon as a subset of children (even one child) is found that have the smallest DeweyIDs related to QN(*IE*). Instead of processing the children of a node from left to right (as done in the AND case), you look at the smallest DeweyID each one provides initially related to QN(*IE*) and sort them according to these DeweyIDs. This results in the order C_{i_1}, …, C_{i_n}. We refer to the mentioned DeweyID of C_{i_j} as d_{i_j}. The balancing process begins from C_{i_1}. Notice that $d_{i_j} \leq d_{i_{j+1}}$. If $d_{i_j} < d_{i_{j+1}}$, it shows that all d_{i_k}, $k \leq j$, are the smallest DeweyIDs among all children. As a result, C_{i_k} s are the balanced children that will participate in the joining process (procedure *ORBalance* in Figure 23). The rest of the process is similar to what we outlined for logical *AND*. For each balanced child, a list of joinable DeweyIDs is formed (lines 123-124 in Figure 23). Members of these lists are joined using a Cartesian product and each result is enriched w.r.t. members of UQN(*IE*).

Processing a logical operator *XOR* is similar to *OR*, but the balancing process needs some modifications. After the balancing process is done similar to logical *OR*, if the number of balanced children is greater than one, the results of these children are skipped. The balancing process will be finished when only one child of *Evaluation Tree* node is the result of balancing. Hence, we will not pay further attention to *XOR* in the rest of this paper.

Lemma 2. S^3.v3 correctly computes all possible matches for a given QTP *Q* with only one join point against XML document *Doc*.

Proof. Suppose that the only join point of *Q* is *JP* and its leaves are L_1, ..., L_n. Considering the algorithm demonstrated in Figure 21, *Q* is traversed from its root. If the root is itself the node *JP*, the root of the *Evaluation Tree* is formed related to the root of *Q* (line 62) and then all branches of *JP* are traversed (lines 63-66). Since these branches do not have any join point, each constructed *Evaluation Tree* related to them has only a single node. On the other hand, if the root is not the node *JP*, then *Q* is traversed to node *JP*. An *Evaluation Tree* node *ER* is formed based on node *JP*. Then, the traversed nodes from the root of *Q* to node *JP* are set as nodes for which *ER* is responsible to produce proper DeweyIDs (lines 39- 60). Traversing *Q* from node *JP* is the same as mentioned before. As a result, the *Evaluation Tree* related to *Q* is the tree *ET* having two levels. The root of *ET* is related to *JP* and its leaves (EL_1, ..., EL_n) are related to the leaves of *Q* (L_1, ..., L_n), respectively.

As depicted in lines 6-17 in Figure 23, the *Evaluation Tree* leaves fetch document nodes based on the match pattern assigned to the *Evaluation Tree*. Then, each leaf (for example EL_1) forms a sub-match containing DeweyIDs related to L_1 and UQN(L_1), based on the DeweyIDs of fetched nodes and using UQN(L_1) (see *upperNodes* list in Figure 21). The list *upperNodes* used in Figure 21 contains node *JP* because, when procedure *getEvalTree* is called recursively, we are at the node that is a join point.

Therefore, when procedure *getEvalTree* is called for a child of the join point, the child is sent to the procedure as the variable *root* and the parent of *root* is the join point itself. As a result, in lines 35-37 in Figure 21, the join-point node is added to the list *upperNodes* of each child of the *Evaluation Tree* node related to the respective join point (here, *JP* is added to *upperNodes* lists of EL_1, …, EL_n).

Considering the above situation, the entire sub-matches that EL_1, ..., EL_n send to *ER* contain a DeweyID related to *JP*. Consequently, during the balancing process in *ER*, all required information is available. *ER* can simply check whether its children are balanced based on its associated logical operator or not (procedures *processAND* and *processOR* in Figure 23). As a result, *ET* is able to produce all required matches related to a given match pattern. In addition, each possible result that matches *Q* belongs to one of the *AND*-QTPs resulting from parsing *Q*. Furthermore, we are able to produce all matches related to each parsed *AND*-QTP *PQ,* because each result matching *PQ* corresponds to one of the match patterns obtained by executing *PQ* against the *Structural Summary* of *Doc* (see Theorem 1 in [13]). Based on the following facts, S^3.v3 correctly computes all possible matches, because each final match belongs to one of the parsed *AND*-QTPs and we are able to produce all matches related to each parsed *AND*-QTP.♣

Notice that, when *AND*-QTPs are executed against the *Structural Summary* of a document, the resulting match patterns differ at least in one member. Therefore, their related matches also differ at least in one member. But when other logical operators are involved in a given QTP *Q*, two or more of the resulting match patterns may be found to be equal w.r.t. *Output(Q)*. Therefore, their related matches may be equal, too, and only one of the equal results is considered as output.

Theorem 2. The method S^3.v3 correctly computes all possible matches for a given QTP *Q* against XML document *Doc*.

Proof. Consider that *ET* is the *Evaluation Tree* constructed based on *Q*. If LP_1, ..., LP_n is the lowest join point in *Q* (there is no other join point between the above nodes and leaves of *Q*), then the sub-trees which have LP_1, ..., LP_n as their roots are trees with a single join point (see Definition 5). Consider a given match pattern *MP*. Based on Lemma 2, using each LP_i and its related subset of *MP*, we are able to produce all required sub-matches related to LP_i*s*. These join points (LP_i*s*) are divided into some groups. Members of each group are descendants of a join point in *Q* (Consider the nearest join point that may be the root or an inner node of *Q*). Suppose that G_1 is one of these groups, which is a descendant of the join point IP_1. In this case, the sub-tree of *Q* with root IP_1 has members of G_1 as its leaves and is a tree with a single join point (see Definition 5). As mentioned before, the *Evaluation Tree* nodes of *ET* related to members of G_1 are able to produce all related sub-matches containing DeweyIDs related to the associated QTP node of members of G_1. Therefore, members of G_1 produce all required DeweyIDs. Since the sub-QTP is bounded to IP_1 and members of G_1, and members of G_1 produce all required DeweyIDs, then, based on Lemma 2, the node of *ET*, which is associated to IP_1, produces all expected sub-matches. Using the same argument, we can show that the root of *ET* produces all possible matches related to the match pattern *MP*. Consequently, with the same argument coined in Lemma 2, we prove that S^3.v3 correctly computes all possible matches for QTPs against XML documents. ♣

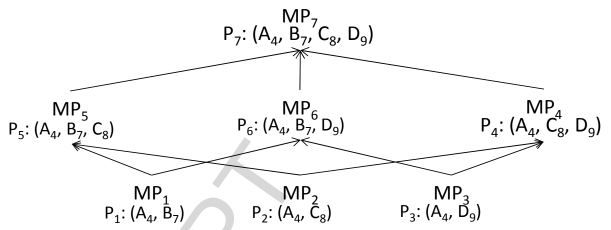


Figure 9. Sample DAG of correlated match patterns related to QTP Q_1 and *Structural Summary* S_1.

4.4 Correlated Match Patterns and Super-Patterns

S^3.v3 still has the problem of redundant I/O and matching processes especially when a given QTP contains logical *OR*. In order to find all document nodes matching a QTP containing *OR*, the QTP should be parsed. If we execute a given QTP against the *Structural Summary* of an XML document without considering the logical *OR* operators, then we lose those match patterns that only match at least one branch of *OR* (not the entire branches). Consequently, final matches related to this kind of match patterns are also lost. As illustrated in Example 4, parsing QTPs is the main source of redundant I/Os as well as matching processes that arise from match patterns having common members. A first idea to solve this problem is to identify those match patterns (called correlated patterns), of which one of them is a subset of another, and produce DAGs of such match patterns. Thus, results of a match pattern can be sent to an upper match pattern (its superset) in the DAG. Final results are gained from DAG nodes that do not have any successor (match patterns without any other superset). Using this strategy enables a substantial improvement. Since some of the match patterns will receive sub-results from different match patterns, they do not have to perform all document accesses and matching processes required.

Example 6. Consider QTP Q_1 and *Structural Summary* S_1 again. Figure 9 shows a correlated DAG of match patterns. An *Evaluation Tree* is formed for each match pattern. *Evaluation Trees*, related to match patterns MP_1, MP_2, and MP_3 (denoted as E_1, E_2 and E_3), fetch document nodes and produce the corresponding results. E_5 receives data from E_1 and E_2 and, therefore, E_5 does not need to fetch anything from the document. E_5 is responsible to join its input results received from E_1 and E_2 and augment its output with additional information. In addition, E_5 has to send those results, which cannot be joined with any other results, to E_7. This task is necessary because, while these kinds of results are not joinable with other results, they match Q_1 w.r.t. E_1 or E_2. For example, (a_2, b_1) and (a_5, b_4) are results produced by E_1. E_5 joins (a_2, b_1) with (a_2, d_2) produced by E_2 and forms (a_2, b_1, d_2), but (a_5, b_4) does not have any counterpart produced by E_2 but matches *Q1* as a final result. Furthermore, E_7 receives results from E_4, E_5, and E_6. It is obvious that E_7 only needs two of these inputs and the third one is completely redundant. ♣

Although forming DAGs of correlated match patterns helps preventing lots of redundant I/O and matching processes as depicted in Example 6, Lemma 3 below provides us with a simple way to optimize the matching process. We exploit the power of Lemma 3 in the next generation of the S^3 algorithms, called S^3.v4:

Lemma 3. Consider a given QTP Q, two match patterns MP_1 and MP_2 and their related *Evaluation Trees* E_2 and E_2. If $MP_1 \subset MP_2$ then matches returned by E_2 cover all matches returned by E_1.

Proof. Let LS_1 be leaves of QTP covered in MP_1 and LS_2 be leaves of QTP covered in MP_2. Because $MP_1 \subset MP_2$ and each member of these match patterns is related to one of the QTP nodes of Q, then the leaves covered in MP_1 are a subset of leaves covered in MP_2 ($LS_1 \subset LS_2$). Let $L_1 \in LS_1$, $L_1 \in LS_2$, $L_2 \in LS_2$, but $L_2 \notin LS_1$. Suppose that JP is the join point of L_1 and L_2 in Q. MP_1 and MP_2 are execution results of Q against the *Structural Summary* of the XML document. As a consequence, both MP_1 and MP_2 match Q. Since $MP_1 \subset MP_2$, we can conclude that all CIDs associated to L_1 and its ancestors (including JP) are the same in MP_1 and MP_2. L_1 and L_2 are descendants of JP and JP is satisfied in both MP_1 and MP_2 while MP_1 does not have any CID related to L_2 and its ancestors up to the JP. Since JP is satisfied once with branch L_2 and once without it, it is clear that JP carries the logical operator *OR*. Therefore, the join points of each leaf in the set ($LS_2 - LS_1$) and all leaves in the set ($LS_2 \cap LS_1$) have a logical *OR* operator.

Because $MP_1 \subset MP_2$, input streams associated to leaves in LS_1 are the same for E_1 and E_2. Therefore, each result like r_1 formed by E_1 will also be formed by the subset of E_2 which corresponds to members of LS_1 as a sub-result. They should not be discarded, because joining these sub-results with sub-results produced by the subset of E_2 (which corresponds to members of ($LS_2 - LS_1$)) have to be checked in join points having logical *OR*. Consequently, if sub-results like r_1 have a joinable counterpart sub-result (corresponding to leaves in ($LS_2 - LS_1$)), they turn into a more complete result. Otherwise, they are not discarded and considered alone as a match, because they are the same as outputs of E_1 which match Q.♣

The main difference between S^3.v3 and S^3.v4 lies in the optimized selection of match patterns done in S^3.v4 to produce the final result. In S^3.v4 prior to using the *Evaluation Tree* to produce matches related to match patterns, DAGs of correlated plans are formed. Those match patterns that do not have any successor (match patterns that do not have any other superset) are selected (called *Super-Patterns*) as candidates to produce the final result.

Theorem 3. The algorithm S^3.v4 computes all possible matches for a given QTP Q against an XML document *Doc*.

Proof. As described in Theorem 2, we can use the *Evaluation Tree E*, constructed w.r.t. Q, to produce all results related to a match pattern. Suppose that m is one of the final results of Q that is not produced by S^3.v4. It is clear that m is related to one of match patterns (consider *MP*) obtained by the execution of parsed *AND*-QTPs against the *Structural Summary* of *Doc*. Each final result has to match one of the parsed *AND*-QTPs. *MP* could not be one of super-patterns selected by S^3.v4, because, based on Theorem 2, match patterns produce the complete results related to them using the *Evaluation Tree*. *MP* is not one of the other match patterns obtained by the execution of parsed *AND*-QTPs against the *Structural Summary* of *Doc*, because each of them is a subset of one of the selected super-patterns and, based on Lemma 3, super-patterns produce the complete results of their correlated patterns. As a consequence, m is not related to any of the above match patterns which is a contradiction. ♣

5. PROCESSING *NOT* OPERATORS

An important logical operator is *NOT*. Due to the flexible structure of XML documents, it becomes highly probable to search document for elements that do **not** have a specific pattern among their descendants. For example in Q_5 below, we search for *A* elements which do not have the pattern (*B[/C]/D*) among their descendants.

Q_5: *//A[NOT(.//B[./C]/D)]*

Definition 6. Consider QTP Q and N as one of Q*'s* nodes. N is a *NOT-Point* if it has a child connected to it by a *NOT* operator. The descendants of N are referred as *NOT-Pattern*.

For example in QTP Q_5, *A* is a *NOT-Point*. Queries may have more than one *NOT-Point*, even in a nested manner, like Q_6 below. In an XML document, *A* elements are considered as final results of Q_6 if they have a *B* element among their descendants and the respective *B* definitely has a *C* element as its child.

Q_6: *//A[NOT(.//B[NOT(./C)])]*

Since those patterns are searched, when logical *NOT* operators are present, that a specific sub-patterns (*NOT-Pattern*) is not associated to them, an idea is to execute the QTP without its *NOT-Pattern* and then execute the entire QTP without considering the *NOT* operator. Therefore, final results are those results of the first execution that are not a subset of a result of the second execution. Processing QTPs having NOT operators also needs parsing the QTP into some *AND*-QTPs. During the parsing of a QTP, when a *NOT-Point* is reached, the QTP is divided into two sections. The first section is made by trimming the QTP at the *NOT-Point*, called *Positive* part. The second part, called *Negative* part, is the cut part of the QTP in the previous step. These two parts are parsed separately. Parsed QTPs related to the positive part together with each parsed QTP related to the negative part are considered as parsed QTPs of the main QTP. After execution of each parsed *AND*-QTP, we achieve some match patterns that match the QTP as a whole, without consideration of *NOT* operators and some match patterns match only the positive part of the QTP. Both kinds of these match patterns are needed. The first one produces results that match the positive part. Some of these results are false positives, with a pattern of nodes in their ancestors that match the negative part of the QTP. These false-positive results are filtered using the second kind of match patterns. However, as explained above, the execution of simple queries at least needs the execution of two parsed *AND*-QTPs, which may have redundant I/O and matching processes.

As described, the key feature of S^3.v4 is to reduce this drawback arising from parsing QTPs needed to support logical *OR* operators. This method is also extensible to support logical NOT operators. We refer to this extended method as S^3.v5. QTP is parsed as mentioned above and super-patterns are selected in the same manner as done in S^3.v4. The process of constructing the *Evaluation Tree* is almost similar to what is done in S^3.v4. The main difference is that during QTP traversal - similar to a join point, if a *NOT-Point* is reached - a new *Evaluation Tree* node is instantiated (procedure *getEvalTree* in Figure 25). Furthermore, the children of an *Evaluation Tree* node are divided into two categories (procedure *addChild* in Figure 26): *NOTChildren* denotes the list of children that are connected to their parent (*NOT-Point*) with a NOT operator, other children are assigned to the list *POSChildren*. The entire children are still accessible as before via the list *children*.

5.1 Evaluation Tree Construction

Note that, in all previous S^3 methods, DeweyIDs related to inner nodes of a QTP are evaluated using DeweyIDs related to QTP leaves. In other words, processing QTPs is done only by extracting a subset of target nodes in the document, which are related to QTP leaves. However, extracting nodes described above does not always suffice to evaluate the final results when a QTP has a *NOT* operator. Some of final results are related to nodes that do not have any descendants related to the *NOT-Pattern* of a query. For example, in Q_5, the entire *A* elements that do not have any *B*, *C*, and *D* elements among their descendants are matches of Q_5. Consequently, when there is not any DeweyID related to QTP leaves (here *C* and *D*), it is not possible to evaluate the DeweyID of such *A* elements.
Now consider the following query:

Q_7: *//A[.//B][NOT(.//C)]*

By Q_7, we search for *A* elements which have a *B* element, but no *C* elements, as their descendants. Therefore, DeweyIDs of related *B* elements, along with DeweyIDs of related *C* elements (leaves of QTP) suffice to evaluate all final results. These examples show that, when a *NOT* operator is involved, it is sometimes required to fetch target nodes related to the *NOT-Point* directly from the document. In this kind of situations, an *Evaluation Tree* node is instantiated to fetch such nodes and is assigned to the *Evaluation Tree* node related to the *NOT-Point* as an ordinary positive child, called *virtual* child. This process is summarized by three cases below:

Case 1: if the *NOT-Point NP* has only one child *NC*, then an *Evaluation Tree* node *N* is instantiated associated to *NP*. Then the *Evaluation Tree NE* related to sub-QTP rooted at *NC* is evaluated and assigned to *N* as a *NOT-Child*. A virtual child *V* is also instantiated to fetch target nodes related to *MP*(*NP*). *V* is set as an ordinary child of *N* (lines 14-19, 29-34 in Figure 25). Now the *NOT-Point* has two children and we consider an *AND* Operator as the applicable logical operator of *N*. Therefore, the procedure *processAND* (see Figure 26) is executed to process *NOT-Points* having only one child and results of *N* are those outputs of *V* that are not produced by *NE*.

Case 2: if the *NOT-Point NP* has more than one child and its children are connected to *NP* with a logical *AND*-operator, then an *Evaluation Tree* node *N* is instantiated associated to *NP*. With respect to each child C_i of *NP*, a related *Evaluation Tree* CE_i is formed. If C_i is a positive child, then CE_i is added to *N* as a positive child; otherwise, it is added as a *NOT*-child. Furthermore, usage of a virtual child is dependent on the condition of child nodes of *NP*. There are two possibilities: If *NP* has at least one ordinary (positive) child, then there is no need to extract target nodes related to *MP*(*NP*), because *NP* has positive branches and final results have to have data for the entire positive branches. Therefore, the DeweyID of *NP* can be evaluated by these branches. However, if *NP* has no positive child, then we have to extract target nodes related to *MP*(*NP*) via a virtual child *V* as an ordinary (positive) child of *NP*, because *NP* elements, that are results of *N*, do not satisfy all children of *NP*. Therefore, we may have no information to evaluate proper results for *N*.

Case 3: if the *NOT-Point NP* has more than one child and its children are connected to *NP* with a logical *OR* operator, then an *Evaluation Tree* node *N* is instantiated associated to *NP*. The *Evaluation Trees* related to children of *NP* are evaluated and assigned to *N*, similar to case 2. In this case, usage of a virtual child related to *MP*(*NP*) is mandatory because, if *NP* even has

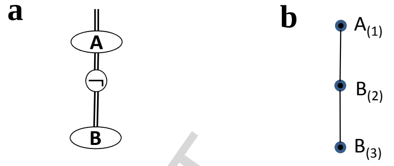


Figure 10. (a) QTP Q_8; (b) *Structural Summary* S_3.

positive children, those *NP* elements that do not satisfy the entire positive children, but satisfy as least one *NOT*-Child, are results of *N*. We may have no information to produce the DeweyID related to *NP* for this kind of results.

In general, when a *NOT-Point* has no positive children or they are connected to it by a logical *OR* operator, it is necessary to extract target nodes related to the *NOT-Point* via a virtual child. The algorithm of constructing the *Evaluation Tree* in S^3.v5 is depicted in Figure 25. Note that, for each leaf of an *Evaluation Tree*, a virtual node is also assigned to provide a uniform method for extracting document nodes. However, these virtual nodes are not considered as a new level (new leaves) in the *Evaluation Tree*.

5.2 Matching Process

Processing *Evaluation Tree ET* begins from the root of *ET* and each node of *ET* (including its root) recursively starts the matching process related to its children (procedure *open* in Figure 26). Each node *N* in the *Evaluation Tree* has one of the following situations:

Case 1: if *N* is associated to a QTP leaf, its results are directly fetched from the document (lines 36-46 in Figure 26).

Case 2: if *N* is associated with a logical *AND* operator, then *N* has at least one positive child. This is because, if QN(*N*) is a *NOT-Point* with one child, then a positive virtual child is added as mentioned before. On the other hand, if *N* has more than one child and all children of *N* are *NOT-Children*, again a positive virtual child is added to *N*. Consequently, we have at least one positive child for nodes like *N*. To evaluate the sub-matches related to *N*, first only positive children are considered and the balancing process is done for them (line 67 in Figure 26). If the positive children get balanced, then, as described in S^3.v3, the lists of joinable nodes related to each child are formed and joined together (lines 73-75 in Figure 26). Results evaluated up to this point are considered as results of *N*, if one of the following conditions exists:

1) If QN(*N*) does not have any *NOT-Child*.
2) If all *NOT-Children* of *N* have no more results and, thus, all of them are removed (lines 76-84 in Figure 26).

If none of the above conditions exists, then the results have to be checked to see whether or not they satisfy existing *NOT-Children*. Satisfying the *NOT-Children* is possible only when all *NOT-Children* are not able to produce DeweyIDs equal to DeweyIDs of evaluated results w.r.t. QN(*N*). Therefore, if at least one of these *NOT-Children* produces such an ID, the evaluated results have to be omitted. To find IDs, described above, *NOT-Children* have to be first balanced using the same balancing process as for the logical operator *OR*. Therefore, we are able to find those children having the smallest DeweyIDs related to QN(*N*). If the DeweyID *NID* (related to QN(*N*)) of balanced *NOT-Children* is smaller than that of the evaluated results of positive children (consider *PID*), then the balancing process is repeated for *NOT-Children* until a *NID* is reached that is greater than or equal to *PID*. If the *NID* reached is equal to the PID, then the evaluated results related to positive children have to be omitted. (procedure *ANDExclude* in Figure 26)

Case 3: If *N* is associated with a logical *OR* operator, then *N* may have a result if at least one of its positive children produces a sub-match. In this case, if all *NOT-Children* do not produce sub-matches with the same DeweyID related to QN(*N*), the result of positive children will not be omitted. On the other hand, if there exists a target node *n* of QN(*N*), such that none of the positive children produce the DeweyID of *n* and at least one of *NOT-Children* or all of them are not able to produce sub-matches having the DeweyID of *n*, then *n* is a result for *N*. In this situation, the children of *N* do not provide any information to produce such an ID. This shows the reason why a virtual child is necessary when an *Evaluation Tree* node has *NOT-Children*. In order to evaluate the sub-matches related to *N*, the balancing process is executed for all children (including the probable virtual child) of *N* (line 20 in Figure 27). After the balancing process is finished, those children are identified whose current result has the smallest DeweyID related to QN(*N*) among all children of *N*. Then a list of joinable nodes related to each balanced child is formed and, finally, these lists are joined together (lines 21-23 in Figure 27). This evaluation is considered producing the results of *N* if:

1) There exists at least one sub-match related to one of the positive children (without considering the probable virtual child) (lines 25-33 in Figure 27). It is not necessary to check the virtual child here, because a target node *n* of QN(*N*) that does not satisfy any child of *N* is not a result of *N*. However, the virtual child participates in the balancing process and produces the DeweyID of *n*, too. Therefore, the virtual child has not to be considered at this point to enable us to omit sub-results that neither satisfy original, positive children of *N* nor *NOT-Children* of *N*.
2) The evaluated results do not have a sub-match related to one of the *NOT-Children* of *N* (lines 34-42 in Figure 27). As mentioned before, when an *Evaluation Tree* node has at least one *NOT-Child*, a virtual positive child is added to it. Thus, if we have a target node *n* that does not satisfy positive children of *N*, but satisfies all *NOT-Children*, then the DeweyID of *n* will be produced in *N* because the virtual child participating in the balancing process provides the DeweyID of *n*.
3) If one of the *NOT-Children* of *N* has no more results, then all evaluated results from this point are definitely results of *N*, because they at least satisfy one child of *N* (the finished *NOT-Child*) (lines 13-18, 24 in Figure 27).

Note that in previous methods like S^3.v4, each output *U* of an *Evaluation Tree ET* is actually a final result for a given QTP *Q* because *U* has proper DeweyIDs w.r.t. nodes of *Q*. However, when a logical *NOT* operator is involved in a QTP, any output of an *Evaluation Tree* is not necessarily a final result.

Example 7. Consider the QTP Q_8 and *Structural Summary* S_3 in Figure 10. Executing Q_8 against S_3 results in the following SMP $\{MP_1(A_1, B_2), MP_2 (A_1, B_3)\}$. Therefore, the *Evaluation Trees* ET_1 and ET_2 are fed by MP_1 and MP_2 , respectively, to evaluate the final results. However, consider *a* as a target node of A_1 which has a *B* element *b* related to B_2 as its child but does not have any *B* elements as its ancestor related to B_3. It is obvious that *a* has at least one *B* element as its descendant (or child) and it cannot match Q_8. However, if we perform the matching process using the above SMP, *a* is considered as a match for Q_8 (a false-positive result) w.r.t. MP_2. The target node *a* is extracted by a virtual child *V* of node *A* in ET_1 and *b* is extracted by node *B*. Node *V* sends the DeweyID of *a* directly to node *A* and node *B* produces the DeweyID of *a* based on MP_1 and sends it to node *A* of ET_1. It is obvious that the DeweyID of *a* is discarded in the *A* node of ET_1. On the other hand, *a* is also extracted in the *Evaluation Tree* ET_2 by a virtual child *V*. Since there is not any *B* element in the target nodes of B_3 as child of *a*, node *A* of ET_2 receives the DeweyID of *a* only from its virtual child and therefore outputs *a* as a match while it is a false-positive result.♣

In fact, when we process a target node *n* of a *NOT-Point N* in a given QTP *Q*, we have to be confident that *n* has no pattern of elements among its descendants that match a sub-QTP rooted at *N*. However, as illustrated in the above example, it is probable that a target node of a *NOT-Point* is processed in two or more separated match patterns and, therefore, related *Evaluation Trees* are not able to verify that their results are a final result or a false-positive one. Consequently, when at least one logical *NOT* operator is involved in a QTP, more investigation is needed.

It is clear that a false-positive result such as f_1 is an output that satisfies at least one of the match patterns such as MP_1 w.r.t. *Output(Q)*. However, there should be at least another match pattern MP_2 such that: for each $q \in Output(Q)$, $MP_1(q)$ is equal to $MP_2(q)$. Now, it is possible to form a false match f_2 using target nodes related to MP_2, which has equal DeweyIDs to those of f_1 related to *Output*(*Q*). With respect to Definition 4, consider that *NList* is a list which contains those members of *Output(Q)* which are not leaves of *Q* and, in addition, none of them have a descendant that is member of *Output(Q)*. If f_2 does not match *Q* (as it is our assumption to have f_1 as a false positive), it means that

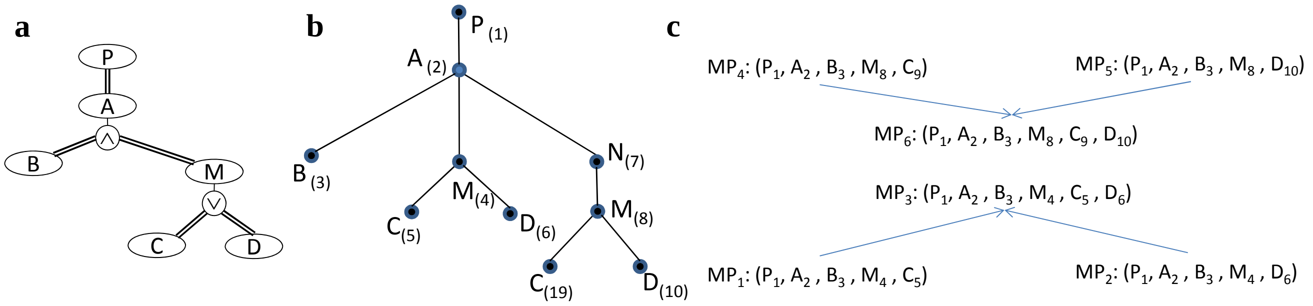

a
P
A
∧
B
M
∨
C
D
b
P (1)
A (2)
B (3)
M(4)
N(7)
M(8)
C(5)
D(6)
C(19)
D(10)
c
MP4: (P1, A2 , B3 , M8 , C9)
MP5: (P1, A2 , B3 , M8 , D10)
MP6: (P1, A2 , B3 , M8 , C9 , D10)
MP3: (P1, A2 , B3 , M4 , C5 , D6)
MP1: (P1, A2 , B3 , M4 , C5)
MP2: (P1, A2 , B3 , M4 , D6)

Figure 11. (a) QTP Q_9; (b) *Structural Summary* S_4; (c) correlated DAGs of match patterns.

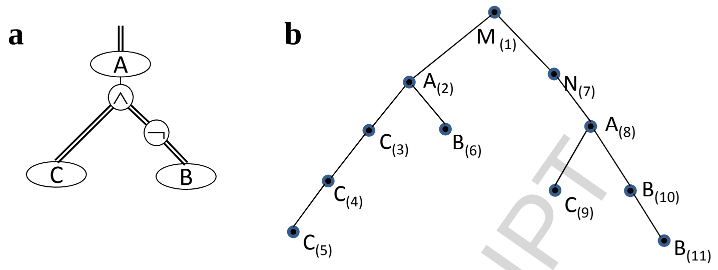


Figure 12. (a) QTP Q_{10}; (b) *Structural Summary* S_5.

f_2 cannot satisfy at least one of QTP nodes listed in *NList* and therefore does not match *Q*. Example 7 depicted a sample instance of this situation. Since f_2 is not formed in the *Evaluation Tree* fed by MP_2, therefore there is not enough information to omit f_1 as a false-positive result. As a result, an expensive procedure should be done to recheck the entire results of *Evaluation Trees* when a QTP involves at least one logical *NOT* operator.

Fortunately, there is an efficient method to deal with false-positive results. In each *NOT-Point*, when a (sub-)match has to be deleted because of *NOT-Children*, it is marked as a dummy match instead of its deletion. Therefore, it is possible to verify outputs of *Evaluation Trees* to remove false-positive results. Those outputs that are equal w.r.t. *Output(Q)* and at least one of them is a dummy output are considered as false-positive results and will be omitted. Consequently, false-positive results are identified without further document access. Note that we did not describe how we deal with false positives in pseudo codes of S^3.v5 (Figure 26 and Figure 27) for sake of simplicity.

Theorem 4. The algorithm S^3.v5 computes all possible matches for a given QTP *Q* against an XML document *Doc*.

Proof. If *Q* does not have a logical *NOT* operator, S^3.v5 behaves the same as S^3.v4. However, if at least one logical *NOT* operator is involved in *Q* as described above, there are three situations for which enough information may not be available to evaluate the DeweyID related to a *NOT-Point*:

1) It has only one child.
2) It is associated with a logical *AND* operator and has no positive child.
3) It is associated with a logical *OR* operator.

All three cases are described above. We showed how *virtual* children are used to obtain enough information to evaluate DeweyIDs related to *NOT-Points*. Furthermore, we described how we deal with false-positive results.♣

6. I/O OPTIMIZATION

As described in previous sections, processing QTPs, containing a logical *OR* or *NOT*, is done by parsing these QTPs into some *AND*-QTPs. The execution of each parsed QTP against the document's *Structural Summary* results in some match patterns that may have common members related to leaves of a given QTP or its *NOT-Points*. This implies that some target nodes are fetched several times due to the separate execution of each match pattern.

In S^3.v4, we proposed the idea of *super-patterns* to get rid of matching processes and results which may also be produced by correlated patterns. Note, all results could be produced using the match pattern that is a superset of all correlated match patterns. However, there are situations that these super-patterns also have common members. Inspired by S^3.v1, it is a good idea to form grouped match patterns (GMPs) of super-patterns by grouping those patterns having common members. S^3.v6 uses this idea.

Example 8. As depicted in Figure 11, we obtain six match patterns by parsing Q_9 and then executing the resulting parsed *AND*-QTPs against the *Structural Summary* S_4. These match patterns can be classified into two DAGs of correlated plans. MP_3 and MP_6 are selected as two super-patterns that are able to produce the final results in S^3.v5. As a consequence, the I/O corresponding to target nodes of *B* with CID 3 is reduced to one third, whereas it is halved for target nodes of *C* and *D* with CIDs 5, 8, 9 and 10. However, super-patterns MP_3 and MP_6 still have common nodes (B_3). Grouping these two super-patterns helps to optimize access to target nodes of B_3. ♣

6.1 Grouping of Super-Patterns

Similar to S^3.v5, a given QTP is first parsed in S^3.v6. Then the resulting *AND*-QTPs are executed against the *Structural Summary* of the document, and super-patterns are identified using DAGs of correlated plans (lines 15-23 Figure 28). Then the list of QTP nodes having direct document access (*LD*) is formed. *LD* is sorted in ascending order based on the distinct occurrences of CIDs related to each member of *LD*. It is clear that grouping super-patterns having the same CID related to *LD*[*0*] produces the minimum number of groups. Furthermore, those super-patterns that may not have a CID related to *LD*[*0*] are classified based on the next member of *LD*. The classification continues until all super-patterns are grouped (lines 1-10 Figure 28).

Evaluation Trees are formed using the main QTP and are fed by grouped super-patterns. These *Evaluation Trees* are constructed in the same way as in S^3.v5. However, each virtual child may be associated to more than one CID because of the existence of different CIDs in a GMP related to a QTP node. Therefore, each virtual child extracts target nodes with different CIDs, but outputs them in a sorted manner (function *groupedStream* used in Figure 29).

Processing of a given *Evaluation Tree ET* begins from its root. Each node in ET recursively achieves sub-matches from its children to produce its related results. The main difference between S^3.v6 and S^3.v5 is that *Evaluation Trees* are associated with a simple match pattern in S^3.v5, but they are associated with a grouped match pattern (GMP) in S^3.v6. Therefore, in S^3.v5, when sub-matches are joined together in a given node *N*, each join result is enriched with DeweyIDs related to ancestors of QN(*N*) up to UQN(*N*), based on the associated match pattern of *ET*. However, *Evaluation Trees* are associated to grouped match patterns in S^3.v6, which probably contain more than one simple match pattern. As a result, ancestors of QN(*N*) probably have different CIDs in the associated GMP of *ET*. Hence, for each group of wrapped match patterns in GMP, having the same CIDs w.r.t. each ancestor of QN(*N*) up to UQN(*N*), one separate result is formed (function *completeToUpperJoinPoint* in Figure 30).

6.2 False-positive Results

Note, although the idea of grouping match patterns in S^3.v6 is inspired by S^3.v1, those false-positive results that are produced in S^3.v1 are not formed in S^3.v6. The following example illustrates this point.

Example 9. Consider the *Structural Summary* S_2 and QTP Q_3 in Figure 5. Execution of Q_3 against S_2 results in the following SMP {$MP_1(E_2, G_4, H_5, K_6, L_7)$, $MP_2(E_2, G_4, H_9, K_{10}, L_{11})$, $MP_3(E_2, G_4, H_{12}, K_{13}, L_{14})$, $MP_4(E_2, G_4, H_9, K_{10}, L_{14})$, $MP_5(E_2, G_4, H_9, K_{13}, L_{11})$, $MP_6(E_2, G_4, H_9, K_{13}, L_{14})$}. As explained in Example 3, grouping the above match patterns results in an SGMP having only one GMP {(E_2, G_4, (H_5, H_9, H_{12}), (K_6, K_{10}, K_{13}), (L_7, L_{11}, L_{14}))}. Therefore, the target nodes of G_4 are extracted only once in S^3.v1 and S^3.v6 during the matching process.

Again consider the following target nodes: *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}*. As explained in Example 3, the joining process in S^3.v1 returns the following three results {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)]}.

We explained in Example 3 why m_1 and m_2 are false-positive results, while they are results of the joining process. One of the advantages of S^3.v6 over S^3.v1 is that false-positive results are not formed during the joining process in S^3.v6 and, therefore, the evaluated results do not need more investigation. Consider *ET* as the *Evaluation Tree* formed based on Q_3 in S^3.v6. The leaf node *G* in *ET* receives *RF(4)* and, based on the above GMP, is responsible to enrich the received sub-matches with a proper DeweyID related to E_2. Therefore, the output of node *G* in *ET* is: {g_1:(1.3, 1.3.5.3), g_2:(1.3, 1.9.5.1)}. The leaf node *K* in *ET* receives *RF*(6), *RF*(10), *RF*(13). Members of *RF*(6) only match MP_1. Members of *RF*(10) match MP_2 and MP_4 which both of them have H_5 in common. Members of *RF*(13) match MP_3, MP_5, and MP_6. With respect to *H* as the ancestor of *K*, MP_3 has H_{12} and MP_5 and MP_6 have H_9 in common. Therefore, corresponding to the each member of *RF*(13), node *K* in *ET* produces two separate sub-matches. In this example, the output of leaf node *K* in *ET* is: { k_1:(1.3.7.1, 1.3.7.1.3), k_2:(1.5.3, 1.5.3.7), k_3:(1.9.3, *1.9.3.5.9.1*), k_4:(*1.9.3.5.9*, *1.9.3.5.9.1*)}. With the same manner, the output of node *L* in *ET* is: { l_1:(1.3.7.7, 1.3.7.7.3), l_2:(1.3.7, 1.3.7.9), l_3:(1.9.3, 1.9.3.5.9.3), l_4:(1.9.3.5.9, 1.9.3.5.9.3)}.

The node *H* in *ET* receives the above sub-matches of nodes *K* and *L* in *ET*. These sub-matches are joined based on their related DeweyID of *H*. Therefore, k_1 and l_1 are not joined ($1.3.7.1 \neq 1.3.7.7$) while 1.3.7.1.3 and 1.3.7.7 are joined in S^3.v1 as part of m_1. Furthermore, k_1 and l_2 are not joined in S^3.v6, too, because they have different DeweyIDs related to *H* whereas 1.3.7.1.3 and 1.3.7.9 are joined in S3.v1 as m_2 while their related CIDs do not match any match pattern in the above SMP. Finally, the output of node *H* in *ET* is {(k_3, l_3), (k_4, l_4)}, where these sub-matches are joined with g_1 and g_2 , respectively, in node *E* of *ET* (its root) to form the final results of Q_3.♣

The above examples illustrated that false-positive results appearing in S^3.v1 are not formed anymore during the matching process in S^3.v6. However, the following example reflects that, if a QTP has a logical *NOT* operator, false-positive results appearing in S^3.v5 also appear in S^3.v6.

Example 10. Consider QTP Q_{10} and *Structural Summary* S_5 (Figure 12). Execution of Q_{10} against S_5 and a subsequent selection of super-patterns results in the following SMP {$MP_1(A_2, B_6, C_3)$, $MP_2(A_2, B_6, C_4)$, $MP_3(A_2, B_6, C_5)$, $MP_4(A_8, B_{10}, C_9)$, $MP_5(A_8, B_{11}, C_9)$}. A quick look over the resulted SMP indicates that it has a match pattern having common members (B_6, C_9). This fact confirms that grouping is helpful to optimize the I/O and matching process. B_6 has the most occurrences in the above SMP. Therefore, leaf *B* is selected to group match patterns. The resulted SGMP is {$GMP_1(A_2, B_6, (C_3, C_4, C_5))$, $GMP_2(A_8, B_{10}, C_9)$, $GMP_3(A_8, B_{11}, C_9)$}. Now consider *a* as a target node of A_8 which has a *B* element *b* related to B_{10} and a *C* element *c* related to C_9 as its children, but does not have any *B* elements as its descendant related to B_{11}. It is obvious that *a* has at least one *B* element as its descendant (child) and it cannot match Q_{10}. However, if we perform the matching process using the above SGMP, (*a*, *c*) is evaluated as a false-positive result. Since *b* and *c* are related to GMP_2, both of them produce the DeweyID of *a* and send it to node *A* in the *Evaluation Tree* ET_2 associated with GMP_2. It is obvious that the DeweyID of *a* is discarded in the *A* node of ET_2. On the other hand, *c* is also extracted in the *Evaluation Tree* ET_3 which is associated to GMP_3 (C_9 is in common between GMP_2 and GMP_3). Since there is not any *B* element in target nodes of B_{11} as child of *a*, node *A* of ET_3 receives the DeweyID of *a* only from its *C* child and outputs (*a*, *c*) as a match, while it is a false-positive result.♣

As illustrated in the above example, false-positive results are produced when match patterns, which must be processed together, are distributed among different GMPs. In Example 10, *A* elements should have no *B* elements in their ancestors to be a match for Q_{10}. Therefore, it is reasonable to process MP_4 and MP_5 together as a GMP. If this happens, then *A* elements that only have a *B* element as their child related to B_{10} are discarded in node *A* of the related *Evaluation Tree* because node *B* has enough information to send a DeweyID of this kind of *A* elements to node *A*. Therefore, a key idea is to delay grouping of super-patterns. After execution of a given QTP *Q* against the *Structural Summary* of a document and selection of super-patterns, the QTP node for which grouping should be done has to be selected. Now the above super-patterns are processed to find the ones that are equal w.r.t. *Output(Q)* and they are grouped. Then these grouped patterns, along with the other super-patterns that are not grouped, are transformed to some GMPs based on the QTP node selected for grouping (function *groupSuperPattern* in Figure 28). Therefore, when super-patterns are grouped, we are sure that those match patterns having the same members related to the entire members of *Output(Q)* are grouped into a single GMP.

Example 11. Consider QTP Q_{10} and *Structural Summary* S_5 again. Output nodes of Q_{10} are nodes *A* and *C*. As can be seen in Example 10, MP_4 and MP_5 are equal w.r.t. Output(Q_{10}). Therefore, first MP_4 and MP_5 are grouped and then the resulting GMP is grouped with MP_1, MP_2, and MP_3. The final SGMP would be {$GMP_1(A_2, B_6, (C_3, C_4, C_5))$, $GMP_2(A_8, (B_{10}, B_{11}), C_9)$}. Now, only a single *Evaluation Tree* related to GMP_2 extracts instance nodes of B_{10} and B_{11}. Therefore, those instance nodes of A_8 which have a *B* element related to B_{10} and a *C* element related to C_9 as their children, but do not have any *B* elements as their descendants related to B_{11} , are omitted and are not considered even as false-positive results. ♣

Lemma 4. Consider a given grouped match pattern *GMP* and an *Evaluation Tree ET* which is formed based on a given QTP *Q* as described in the algorithm S^3.v6. If *Q* does not have any logical *NOT* operator, then *ET* produces all results related to *GMP* without producing false-positive results.

Proof. Consider *E* as a node of *ET*. If *E* obtains its required input directly from the document (via a virtual child), its related outputs which are formed by applying function *completeToUpperJoinPoint* (see Figure 30) are sorted w.r.t. *UQN(E)* to *UQ(E),* because virtual children sort their extracted document nodes based on their DeweyID. On the other hand, if *E* is an inner node (or root) of *ET*, *E* produces its output using a Cartesian product. Therefore, if inputs of *E* are sorted w.r.t. *QN(E)*, *E* is able to join the entire inputs having the same DeweyID related to *QN(E)*. The entire join results are kept sorted after applying function *completeToUpperJoinPoint* w.r.t. *UQN(E)* to *QN(E)* (line 33 in Figure 30). It is straightforward to deduce that inputs of *E* are sorted w.r.t. *QN(E)*, because these inputs are the result of some nodes of *ET* that directly or indirectly (recursively) achieve their inputs via virtual children. Therefore, the root of *ET* is able to produce all results related to *GMP*. To show that none of the above results are false positives, consider MP_1, $MP_2 \in GMP$ and *E* as a node of *ET* having at least two children *B* and *C*. Also, consider B_1 as output of node *B* w.r.t. MP_1, C_1 as output of node *C* w.r.t. MP_1, B_2 as output of node *B* w.r.t. MP_2, C_2 as output of node *C* w.r.t. MP_2.

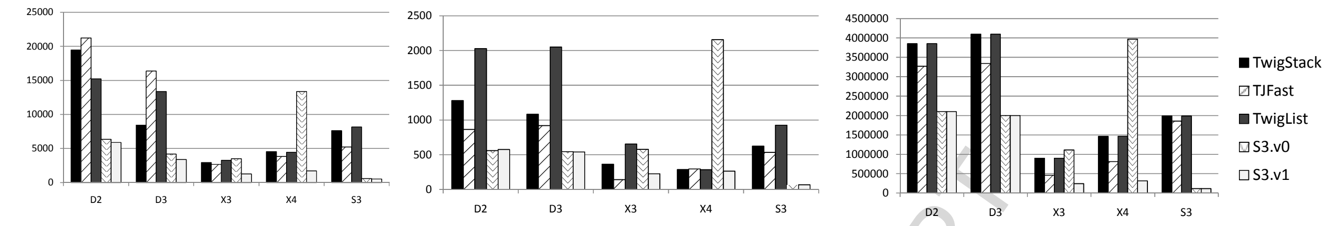
It is clear that, if two sub-results of B_1 and C_1 join w.r.t. to the logical operator associated to *E*, the result is an output for *E* w.r.t. MP_1. We have the same story for sub-results of B_2 and C_2. However, if a sub-result of B_1 joins with a sub-result of C_2, the result may be a false positive. If $MP_1(A)$ does not have any CID in common with $MP_2(A)$, then it is not possible to have a join between sub-results of B_1 and C_2 because CIDs and, therefore, DeweyIDs related to *E* which produced by node *B* definitely differ from those produced by node *C*. However, if $MP_1(A)$ has at least one CID in common with $MP_2(A)$, then it is probable that both B_1 and C_2 produce sub-results having the same DeweyID related to *E*. Therefore, such sub-results join in node *E* and clearly satisfy sub-QTP rooted at *QN(E)*. It is obvious that results like *ER*, formed by joining sub-results of B_1 and C_2, belong to a match pattern other than MP_1 or MP_2, say MP_3. Furthermore, *ER* will be involved in a final result *FR* in *ET*. If MP_3 has been grouped in a grouped match pattern other than *GMP*, say GMP_2, *FR* is also produced in the *Evaluation Tree* ET_2 related to GMP_2. On the other hand, results related to *GMP* and GMP_2 have to be different at least w.r.t. to the leaf *G* for which the grouping is done (each match pattern has a distinct CID related to *G*). However, *FR* is produced in both *ET* and ET_2, and this is a contradiction; therefore, MP_3 has to be grouped into *GMP*. Hence, *ET* produced all results related to *GMP* without producing false positives.

Lemma 5. Consider a given grouped match pattern *GMP* and an *Evaluation Tree ET* which is formed based on a given QTP *Q* as described in the algorithm S^3.v6. If *Q* has at least one logical *NOT* operator, then *ET* produces all results related to *GMP* without producing false-positive results.

Proof. With respect to the rule of grouping in S^3.v6, the entire match patterns having the same members related to the entire members of *Output(Q)* are grouped into a single grouped match pattern. Consider $MP_i \in GMP$ is a set of match patterns having the same members related to all members of *Output(Q)*. Furthermore, consider that *P* is a set of target nodes that, for each member p_j of *P*, there exists a QTP node $q \in$ *Output(Q)* such that p_j is a target node of *q*. If *P* satisfies *Q* without considering the non-output nodes of *Q*, but does not satisfy *Q* as a whole because of some MP_is like MP_N (by considering descendants of its members), then *P* is not a final match for *Q*. It is obvious that *P* is not able to satisfy at least one of the *NOT-Points* like $NP \in$ *Output(Q)*. Node *NP* occurs in one of the following situations:

1) *NP* has only a single child: In this situation, node *ET(NP)*[3] obtains sub-results having DeweyIDs equal to *P(NP)* w.r.t. MP_N. Since *ET(NP)* achieves all instance nodes of $(MP_N(NP))$ via its associated virtual child, all results having DeweyID equal to *P(NP)* are omitted in *ET(NP)* and, therefore, there is not any possibility to produce *P* (see procedure *processAND* in Figure 30 as it is used for *NOT-Points* having only a single child).
2) The associated logical operator of *NP* is a logical *AND* and all children of *NP* are *NOT-Children*: In this situation, at least one of *NOT-Children* produces a sub-result having a DeweyID equal to *P(NP)*. Therefore, the same *NOT-Child* produces a sub-result having a DeweyID equal to *P(NP)* in *ET* which leads to the omission of the DeweyID equal to *P(NP)* extracted by virtual child associated to node *NP* in *ET*. As a result, there is no possibility to form *P* in this situation (procedure *processAND* in Figure 30).
3) The associated logical operator of *NP* is a logical *AND* and *NP* has at least one positive child: If, w.r.t MP_N, at least one of the *NOT-Children* produces a sub-result having a DeweyID equal to *P(NP)*, then, as described in the previous situation, there is no possibility to form *P*. Otherwise, if none of the *NOT-Children* produces a sub-result having a DeweyID equal to *P(NP)* w.r.t. MP_N, then one of positive children of *NP* like *PC* has to be the reason that *P* does not satisfy MP_N. Therefore, we have the following cases w.r.t. the sub-QTP *SQ* which is rooted at *QN(PC)*:
   a) If there exists no logical *NOT* operator in *SQ*: then all members of *SQ* are members of *Output(Q)*. Therefore, those members of each MP_i related to *SQ* are the same in all MP_is. As a result, since node *PC* in *ET* is not able to produce a sub-result having a DeweyID equal to *P(NP)* w.r.t. MP_N, other MP_is are not able to produce such sub-results, too. Further, false-positive results are not produced in *ET(PC)* (Lemma 4). Hence, there is no possibility to form *P* in this case.
   b) If there exist only *NOT-Points* like NP_{L1} in *SQ*, such that NP_{L1} does not have any other *NOT-Point* in its descendants: It is clear that children of *ET(PC)* are divided into two groups: 1) a child C_1 that does not have any *NOT-Point* in its descendants and, therefore, C_1 and all descendants are members of *Output(Q)*. As described before, sub-results formed by C_1 are actual results (not false-positive ones), which are the same w.r.t. all MP_is. 2) a child C_2 that has a node N_2 among its descendants which is connected to its parent PN_2 by a logical *NOT* operator. If PN_2 is the node that

[3] If *ET* is an *Evaluation Tree* and q is a QTP node of a given QTP *Q*, *ET(q)* is a node of *ET* which is associated to *q*.



(a) Execution time (ms) (b) I/O time (ms) (c) Number of elements read

Figure 13. Summary of experimental results reported in [13].

is the cause of preventing *PC* and, therefore, *NP* to produce a sub-result s_1 to form P, then it means that N_2 produces a sub-result like ns_1 that omits s_1 in PN_2. It is obvious that ns_1 is formed in *ET* w.r.t. MP_N and therefore omits all results produced w.r.t. all MP_is having the same DeweyID as s_1 related to $QN(PN_2)$. Since sub-results like s_1 are not formed in PN_2 at all, then *PC* and, therefore, *NP* are not able to produce such sub-results to form *P*. Consequently, if there exists a match pattern MP_N that it is not possible to produce DeweyID *P(NP)* w.r.t. it, then the above DeweyID is not formed in *ET* considering all MP_is and, therefore, *P* is not produced as a false positive.

c) If there exist only *NOT-Points* in *SQ* that, in each of their branches, only one *NOT-Point* at most could be found as their descendant: it is possible to show that DeweyID *P(NP)* is not produced in *ET(NP)* in the same way as described in (b). Consequently, it can be recursively deduce that, if nodes *NP* have different levels of nested *NOT-Points*, they also do not produce DeweyID *P(NP)*.

4) The associated logical operator of *NP* is a logical *OR*: With respect to MP_N, all positive children of *NP* do not produce any sub-results having DeweyID *P(NP)* and all *NOT-Children* of *NP* produce at least one sub-result having DeweyID *P(NP)*. Therefore, considering all MP_is, *NOT-Children* of *NP* produce sub-results having DeweyID *P(NP)*. Hence, *ET(NP)* is not satisfied by all its *NOT-Children*. Now consider one of the positive children of *NP* also as *PC*. As stated in (3), we can show when DeweyID *P(NP)* is not formed in *ET(NP)* w.r.t. MP_N to form *P*, it is not produced by all MP_is, too.

Considering the above discussion, we can conclude that, if *P* w.r.t. its descendants in the document cannot satisfy some of the MP_is, then it is not formed in the *ET*, even by considering all MP_is too. It is obvious that no grouped match pattern other than *GMP* is able to produce *P* because those match patterns having a possibility to produce *P* have equal members w.r.t. all members of *Output(Q)* and these match patterns have to be grouped in *GMP* w.r.t. to grouping rules of S^3.v6 (line 4 in Figure 28). Consequently, all nodes of *ET* (including its root) derive their expected sub-results and produce all results corresponding to *GMP* members without producing any false positives. ♣

Theorem 5. The algorithm S^3.v6 computes all possible matches for a given QTP *Q* against an XML document *Doc*.

Proof. During the grouping process of match patterns in S^3.v6, none of the match patterns is deleted and each of them is grouped in a GMP. Therefore, all possible classes of final results are considered. Furthermore, all target nodes related to match patterns grouped in GMP are fetched from the document via virtual children. With respect to Lemma 5, each *Evaluation Tree* produces all results related to its associated GMP and false-positive results are not produced during the matching process. Consequently, we can conclude that S^3.v6 computes all possible matches for a given QTP *Q* against an XML document *Doc*. ♣

7. EXPERIMENTAL RESULTS

7.1 Experimental Setup

We did exhaustive experiments in our previous paper [13] to measure the performance of our methods S^3.v0 and S^3.v1 in comparison with the following well-known QTP processing methods: *TwigStack* [4], *TJFast* [20], and *TwigList* [22]. Figure 13 presents a summary of the experimental results reported in our previous paper [13], which clearly confirm the superiority of our methods (The queries used in Figure 13 can be found in Table 2). In [13], we have used QTPs having different features in our experiments: QTPs having only a single path, as well as some shallow and some deep QTPs. Furthermore, our scalability analysis revealed that the execution time for our competing methods linearly increased with document size. In contrast, the growth rate of the execution time of our methods is less than others as represented in our experiments [13]. On the other hand, our scalability analysis further confirmed that our methods S^3.v0 and S^3.v1 operate using minimum possible memory while the competing methods need more memory to reach their optimum performance.

In this paper, our goal is a cross-comparison of the proposed methods for *AND*-QTPs. Moreover, we compare our methods to trace the improvement of their performance, when processing QTPs using the logical operators *OR* and *NOT*. We have

Table 1. Characteristics of XML datasets used

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

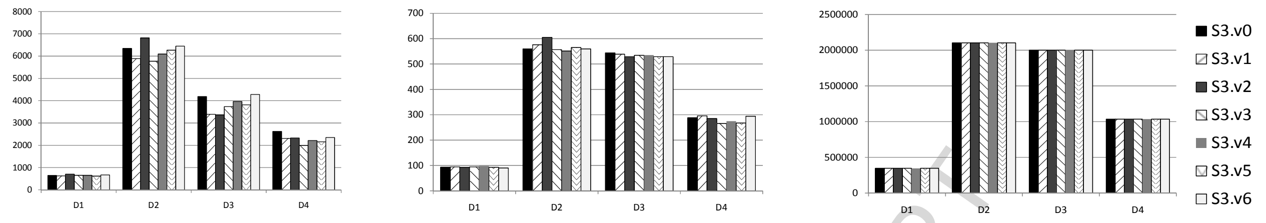
Table 2. Queries used in the experiments

Name	Query	Dataset	Matches
D_1	//article/title	DBLP	346554
D_2	/dblp/inproceedings[title]/author	DBLP	1519938
D_3	/dblp/inproceedings[.//cite/label][title]//author	DBLP	132902
D_4	//article[.//mdate][.//volume][.//cite]//journal	DBLP	47323
D_5	//inproceedings[(.//pages) OR (.//crossref) OR (.//title//sub)]	DBLP	575316
D_6	//inproceedings//title[(.//sub) OR (.//i)]	DBLP	2322
D_7	//article[(.//volume) OR (.//cite) OR (//journal)]	DBLP	346333
D_8	// article //title[NOT(.//sub)]	DBLP	344620
D_9	//inproceedings[.//title[NOT(.//sup/i)][NOT(.//tt)]][//cite/label]//booktitle	DBLP	55718
D_{10}	//inproceedings//title[[NOT(.//sub)] OR [NOT(.//i)]]	DBLP	582563
X_1	/site/regions//item/location	XMark	108750
X_2	//people//person[.//address/zipcode]/profile/education	XMark	15859
X_3	//item[location]/description//keyword	XMark	136282
X_4	//item[location][.//mailbox/mail//emph]/description//keyword	XMark	86533
X_5	//item[location][quantity][//keyword]/name	XMark	207632
X_6	//item//description[(.//text//bold) OR (.//parlist//emph)]	XMark	62162
X_7	//item[(.//location) OR (.//quantity) OR (//parlist//keyword)]	XMark	108750
X_8	//item//description//text[NOT(.//emph)]	XMark	117806
X_9	//item[[NOT(.//keyword)] OR [NOT(.//location)] OR [(.//shipping)]]	XMark	217500
X_{10}	//item[[(.//shipping)][NOT(.//description[NOT(.//keyword)]]]	XMark	58596
N_1	//revisions[//year][//para]//creator	Nasa	1043
N_2	//tableHead[./tableLinks/tableLink/title]//fields/field[definition]/name	Nasa	103380
N_3	//dataset[(.//para) OR (.//heading)]	Nasa	2435
N_4	//definition[(.//footnote) OR (.//para)]	Nasa	11586
N_5	//dataset//description[NOT(.//para)]	Nasa	116
S_1	//Entry//PIR[prim_id][sec_id]	SwissProt	30427
S_2	//Entry/Features[/DISULFID[/from][/to]/Descr][/CHAIN[/from][/to]/Descr]	SwissProt	22437
S_3	//Entry[(.//Ref/Author) OR (.//Keyword)]	SwissProt	50000
S_4	//Entry[mtype][NOT(.//Mod)][NOT(.//Descr)]	SwissProt	0

implemented our methods using our native, Java-based XML database management system, called XTC [24]. The system configuration used for all experiments is as follows: Java 1.6.0_14-b08, Dell® Latitude™ E6500 laptop having Intel® Core™2 Duo CPU P9500 (2.53 GHz), 4 GB main memory, 250 GB hard disk, running Windows Vista™ Business. We have used the well-known datasets DBLP [19], XMark [23] with scaling factor 5, Nasa [25], and SwissProt [25] to verify the robustness, scalability, and structure insensitivity of our methods. Characteristics of our selected datasets are shown in Table 1. The size of the datasets listed is their size in plain text format. The number of nodes along with maximum and average depth of the datasets is computed from the physical representation of these datasets in XTC. For each dataset, different QTPs are chosen representing different features: some of the QTPs are single-path queries, some are shallow, and some are deep. *AND*-QTPs are selected from QTPs used in [13] to compare the performance of S^3.v2 to S^3.v6 against S^3.v0 and S^3.v1. The set of queries used in our experiments are listed in Table 2. Our experiments include cross-comparison of our methods based on their ability to process logical operators in terms of execution time, I/O time, and number of elements read. Furthermore, we have performed a scalability analysis in terms of document size and memory available. Execution time is the time elapsed between the arrival of a QTP in XTC and the delivery of the complete result to the user. Number of nodes read represents the amount of document nodes (in its physical storage format in XTC) that are physically accessed and I/O time is the time spent to perform document access fetching the requested nodes.

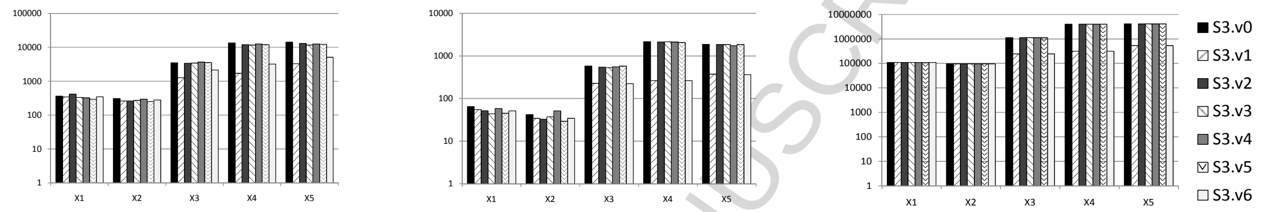
7.2 *Cross-Comparison*

All methods proposed (S^3.v0 to S^3.v6) are able to process *AND*-QTPs; therefore, it is reasonable to compare their performance for evaluating this kind of QTPs. S^3.v0 uses the concepts *QueryGuide* and DeweyID to perform a focused document access resulting in significant performance. Since some match patterns used for QTP processing have common members, separate execution of each of them leads to repeated access of distinct document nodes, which is in contrast to our optimization goal. S^3.v1 attempts to group such match patterns into GMPs to avoid such redundant document accesses. S^3.v2 is a simple extension of S^3.v0 to support the logical *OR* operator. QTPs containing *OR* are parsed into some *AND*-QTPs, which are separately executed; the intermediate results are subsequently merged to prepare the final answer. S^3.v3 utilizes an *Evaluation*



(a) Execution time (ms) (b) I/O time (ms) (c) Number of elements read

Figure 14. Experimental results for *AND*-QTPs against *DBLP*.



(a) Execution time (ms) (b) I/O time (ms) (c) Number of elements read

Figure 15. Experimental results for *AND*-QTPs against *XMark* (scale 5) .

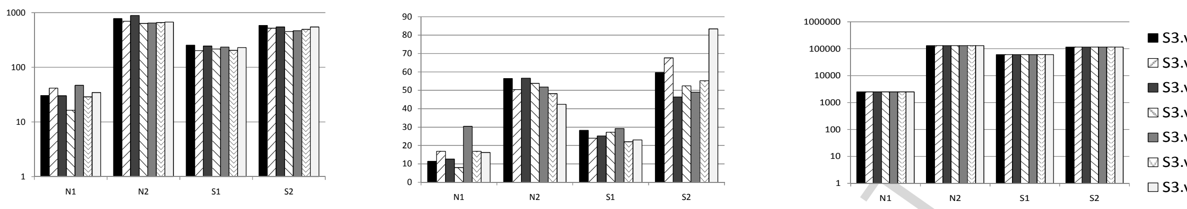
Tree for QTP matching, which provides efficient support of the logical operators *OR* and *NOT*. Because QTP parsing leads to match patterns, which are frequently overlapping and, therefore, causing redundant I/Os, S^3.v4 uses the concept "super-pattern" to get rid of those match patterns which are covered by a super-pattern. S^3.v5 provides support of the logical operator *NOT*. Additionally in S^3.v6, inspired by the idea leading to S^3.v1, super-patterns that still have some overlaps are grouped to optimize I/O and execution time of QTP processing.

AND-QTPs: We chose shallow QTPs in [13], to compare our methods (S^3.v0 and S^3.v1) against *TwigStack* and *TwigList,* because the I/O needed by these methods is comparable to ours. QTPs having several leaves (three or more) are selected to challenge *TwigList,* because it circumvents the time-consuming merging phase burdening *TwigStack* and others. Figure 14 shows our experimental results for four *AND*-QTPs against the DBLP dataset. Obviously, our methods roughly deliver the same performance for *AND*-QTPs. Therefore, methods S^3.v2 to S^3.v6 are also proper solutions for evaluating *AND*-QTPs. Our experimental results for *AND*-QTPs against XMark (scale 5) (Figure 15), Nasa, and SwissProt (Figure 16) also confirm their efficiency and comparability. It is worth noting for QTPs such as X_1, X_2, and X_3 in Figure 15 that the runtime differences of S^3.v1 and S^3.v6 are caused by the grouping mechanism used by them.

OR-QTPs: Our methods S^3.v2 to S^3.v6 are able to process QTPs containing the logical *OR* operator. Figure 17 shows our experimental results for three QTPs using *OR* operators. Parsing QTP D_7 leads to seven *AND*-QTPs to be executed. Because a super-pattern can be exploited, S^3.v4 to S^3.v6 have lower I/O cost and better performance. Moreover, D_5 leads to seven *AND*-QTPs and finally eleven match patterns, where two of them are selected as super-patterns and, in turn, are grouped into a single GMP in S^3.v6. As a consequence, S^3.v6 has the best performance and lowest I/O cost for D_5 (see Figure 17). QTP X_6 clearly reveals the effect of the grouping idea. Execution of X_6 leads to three *AND*-QTPs and finally 594 match patterns which are reduced to 486 super-patterns. Using S^3.v6, these super-patterns turn into 6 GMPs. Our experiments show that S^3.v6 executes X_6 about three to ten times faster than other methods and requires about an order of magnitude less I/O cost. The efficiency of grouping used in S^3.v6 can also be observed in the remaining QTP executions.

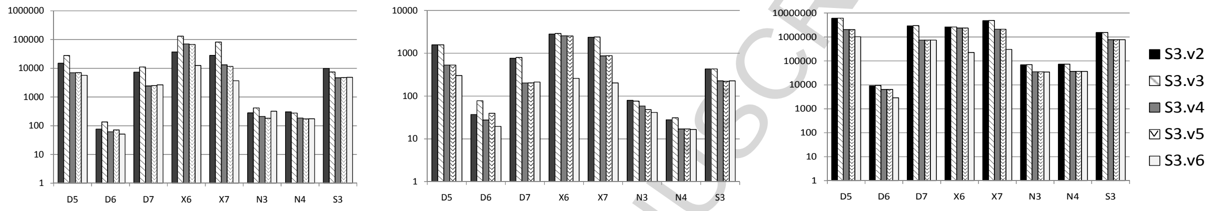
NOT-QTPs: S_3.v5 and S_3.v6 are methods to process QTPs containing logical *NOT* operators. Figure 18 shows the comparison of S^3.v5 and S^3.v6 for three QTPs. Obviously, grouping of match patterns has a major effect on the performance of QTP evaluation. For example, compared to S3.v5, S^3.v6 runs D_{10} seven times faster thereby reducing the I/O cost about six times. We gained the same improvement for X_{10} against the XMark dataset. Experimental results related to S_4 also confirm a runtime and I/O reduction of about an order of magnitude.

The methods *GTwigMerge*, *PathStack¬*, *TwigStackList¬*, and *AllTwigMerge* are dropped from our cross-comparison, because they are all based on *TwigStack*. These methods read the entire instance nodes related to the QTP and, therefore, cause the same I/O as *TwigStack* causes for the *AND*-QTP with the same tree structure (henceforth referred as *AND*-VIEW of a QTP). Furthermore, processing of a QTP containing logical operators like *OR* and *NOT* is more expensive than its *AND*-VIEW because of the inherent overhead of logical OR and *NOT* operators. On the other hand, using the super-pattern concept, we are sure that redundant I/Os are avoided and I/O cost is optimal, because only potential target nodes of QTP leaves are extracted by S^3 methods. Furthermore, the processing time using the *Evaluation Tree* is dependent to the tree structure and the number of nodes read instead of the logical operators related to the nodes in the tree (As can be seen in the Figure 23, the execution time order of *processAND* and *processOR* procedures are similar). Therefore, the execution time of a QTP is comparable to its *AND*-VIEW in our proposed methods. Finally, as reported in [13] and illustrated in Figure 13, our methods S^3.v0 and especially S^3.v1 clearly outperform *TwigStack* for *AND*-QTPs and, as can be seen, our methods have similar performance for *AND*-QTPs (Figure 14).



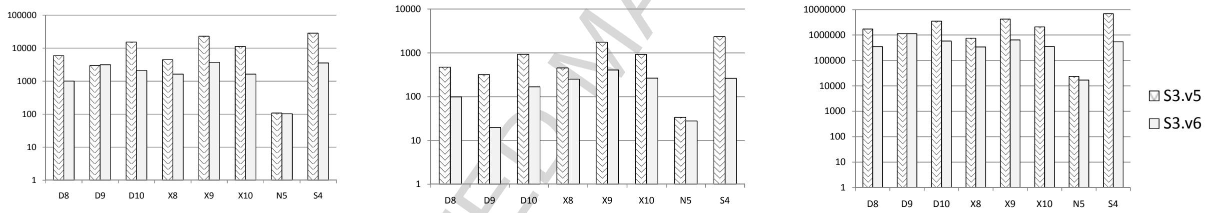
(a) Execution time (ms) (b) I/O time (ms) (c) Number of elements read

Figure 16. Experimental results for *AND*-QTPs against *Nasa* and *SwissProt*.



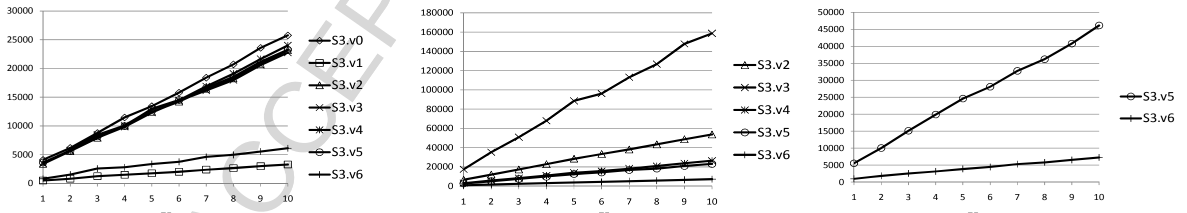
(a) Execution time (ms) (b) I/O time (ms) (c) Number of elements read

Figure 17. Experimental results for QTPs using logical *OR* against *DBLP*, XMark(scale 5), *Nasa,* and *SwissProt*.



(a) Execution time (ms) (b) I/O time (ms) (c) Number of elements read

Figure 18. Experimental results for QTPs using logical *NOT* against *DBLP*, XMark(scale 5), *Nasa,* and *SwissProt*.



X_4 X_7 X_9

Figure 19. Scalability of document size: execution time (ms).

7.3 *Scalability Analysis*

In addition to the cross-comparison of our methods, scalability experiments for document size and memory available (maximum heap size of the Java Virtual Machine) were performed. Earlier results [13] showed that increasing the document size caused a sub-linear growth of the execution time for S^3.v0 and S^3.v1, whereas *TwigStack*, *TJFast*, and *TwigList* embodied linear behavior.

To explore the size scalability of our methods, we chose 10 XMark datasets with a scaling range from 1 to 10. Figure 19 visualizes the scalability characteristics for X_4, X_7, and X_9, respectively. Our experiments show that the growth rate of the execution time of our methods is linear as depicted in Figure 19. The effect of grouping match patterns is also illustrated in Figure 19. For example, execution of X_4 leads to 162 match patterns where most of them overlap. Therefore, only the use of methods S^3.v1 or S^3.v6 is adequate, because the resulting match patterns turn into six GMPs.

We have also checked the scalability of our methods in terms of document size for QTPs having logical operators *OR* or *NOT*. Figure 19 clearly shows that our methods (except S^3.v2) follow sub-linear behavior. Execution of X_7 leads into 54 match patterns, which turn into 6 GMPs in S^3.v6. Our experiments for X_9 also confirm the major effect of grouping on S^3.v6 scalability.

The remaining set of experiments focuses on scalability w.r.t. memory available. Earlier results showed [13] that the performance of *TwigStack*, *TJFast*, and *TwigList* is more dependent on the amount of memory available, because they produce much larger intermediate results than S^3.v0 and S^3.v1.

To check the memory effect, we have chosen four queries D_2, X_5, X_7, and X_9. Increase of memory size does not have a significant effect on the runtime of our methods (see Figure 20), which confirms their effective use of memory. Therefore, our methods behave efficiently when memory size is limited or shared with other transactions in real multi-user environments.

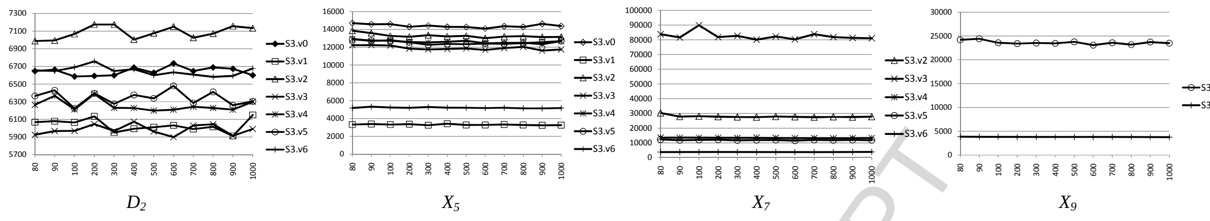


Figure 20. Scalability of memory (MB) available: execution time (ms).

8. Conclusions

We first reviewed some basic definitions and concepts from our previous work proposed in [13]. Our methods are founded upon two critical concepts: DeweyIDs and *QueryGuide*. We use DeweyIDs to label document nodes. These IDs contain all information about their own ancestors. Therefore, during QTP matching, all information related to ancestors of nodes accessed is available without further document access. Execution of a QTP against the *Structural Summary* of a document leads to an SMP, which helps us perform focused document access.

We use the *Evaluation Tree* as a processing structure formed by QTP join points and its leaves to process match patterns. These patterns are results of QTP executions against the *Structural Summary* of the document. The main advantage of *Evaluation Trees* is the ease of evaluating join results. Therefore, use of *Evaluation Tree* is a way to support logical operators other than *AND*. We make parsing of a QTP into some *AND*-QTPs efficient and use it as the base of our methods. We initially introduced S^3.v2 and S^3.v3, but they suffer from redundant I/O and matching process caused by overlapping match patterns. Then, we developed the concept of super-pattern to solve this problem in S^3.v4. We showed that a superset of match patterns covers all matches related to its sub-patterns. Therefore, the selection of super-patterns in S^3.v4 largely reduces the I/O volume during QTP matching and makes QTP parsing an efficient solution.

Using the strength of *Evaluation Trees*, we support the logical operator *NOT* in S^3.v5. The main drawback of S^3.v5 are false-positive results occurring during the matching process and caused by a lack of information in *NOT-Points*. Hence, we had to find and remove false positives from the results in S^3.v5 by keeping results that have to be omitted in *NOT-Points* as dummy results. We implemented the idea of grouping match patterns as our final method in this paper. We reached two goals in S^3.v6:
1) We prevented this method from producing false-positive results. This is done by grouping those match patterns together having enough information to eliminate false-positive results.
2) We reduced a considerable volume of I/O caused by super-patterns having overlaps.

Our experiments in [13] showed the superiority of our methods compared to *Structural Join*, *TwigStack*, *TJFast*, and *TwigList*. Our experiments show that all of our new methods have almost the same performance to process *AND*-QTPs. Moreover, the effect of using super-patterns and grouping idea is observable in our experiments. Furthermore, our scalability tests show that all our methods perform well, even if the size of memory available is limited. In addition, the growth rate of the execution time of our methods is linear when document size increases.

9. 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: Proc. of ICDE Conf., 2002, 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] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, J. Simeon, XQuery 1.0: An XML Query Language, W3C Working Draft, 2007. <http://www.w3.org/TR/xquery>

[4] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: Proc. of SIGMOD Conference, 2002, pp. 310–321.

[5] D. Che, Holistically processing XML twig queries with AND, OR, and NOT predicates, in: Proc. of 2nd Int. Conference on Scalable Information Systems, 2007, Article No.: 53.

[6] 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: Proc. of VLDB Conference, 2006, pp. 283–294.

[7] T. Chen, T.W. Ling, C.Y. Chan, Prefix path streaming: a new clustering method for optimal holistic XML twig pattern matching, in: Proc. of DEXA Conference, 2004, pp. 801–810.

[8] T. Chen, J. Lu, T. Ling, On boosting holism in XML twig pattern matching using structural indexing techniques, in: Proc. of SIGMOD Conference, 2005, pp. 455–466.

[9] S.-Y. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, C. Zaniolo, Efficient structural joins on indexed XML, in: Proc. of VLDB Conference, 2002, pp. 263–274.

[10] M. Dewey, Dewey decimal classification system. <http://www.mtsu.edu/vvesper/dewey.html>.

[11] M. Fontoura, V. Josifovski, E. Shekita, B. Yang, Optimizing cursor movement in holistic twig joins, in: Proc. of CIKM Conference, 2005, pp. 784–791.

[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] S.K. Izadi, T. Härder, M. S. Haghjoo, S^3: Evaluation of tree-pattern XML queries supported by structural summaries, Data and Knowledge Engineering 68(1) (2009) 126-145.

[14] H.V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J.M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, TIMBER: A native XML database, VLDB Journal 11(4) (2002) 274-291.

[15] H. Jiang, H. Lu, W. Wang, B.C. Ooi, XR-tree: indexing XML data for efficient structural joins, in: Proc. of ICDE Conference, 2003, pp. 253–264.

[16] H. Jiang, W. Wang, H. Lu, J. Xu Yu, Holistic twig joins on indexed XML documents, in: Proc. of VLDB Conference, 2003, pp. 273–284

[17] H. Jiang, W. Wang, H. Lu, J. Xu Yu, Efficient processing of XML twig queries with OR-predicates, in: Proc. of SIGMOD Conference, 2004, pp. 59-70.

[18] E. Jiao, T.W. Ling, C.Y. Chan, Pathstack¬: A holistic path join algorithm for path query with not-predicates on XML data, in: Proc. of DASFAA Conference, 2005, pp. 113-124

[19] M. Ley, DBLP Computer Science Bibliography. <http://dblp.uni-trier.de/xml/dblp.xml> (accessed 10.10.07).

[20] 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: Proc. of VLDB Conference, 2005, pp. 193–204.

[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: Proc. of DASFAA Conf., 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: Proc. of 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/>.

[26] T. Yu, T.W. Ling, J. Lu, TwigStackList¬: A Holistic Twig Join Algorithm for Twig Query with Not-predicates on XML Data, in: Proc. of ICDE Conference, 2005, pp. 141-152.

Appendix A

In this appendix, we present pseudo codes of our methods S^3.v3, S^3.v4, S^3.v5, and S^3.v6.

```
class S3-v3
constructor(Q as QueryTreePattern, Doc as XTS)
  this.Q = Q;
  this.Doc = Doc;

procedure execute()
  let parsedQTPs be list of AND-QTPs resulted by
  parsing Q;
  for each $PQ_i$ in ParsedQTPs do
    let $SMP_i$ be execution result of $PQ_i$ against the
  structural summary of Doc;
    for each $MP_j$ in $SMP_i$ do
      ET = getEvalTree($PQ_i$, $PQ_i$.root, $MP_i$, Doc);
      this.ETList.add(ET);
    end for;
  end for;
  for each $ET_i$ in ETList do
    $ET_i$.open();
    if $ET_i$.isFinished
      ETList.remove($ET_i$);
  end for;
  while (true) do
    min = nextMatch();
    if (min = null)
      break;
    else
      output min;
  end while;

function nextMatch()
  if (this.ETList.size() = 0) return null;
  let min be the minimum this.ETList[i].head and
  minIndex be its index in the ETList;
  ETList[minIndex].next();
  if (ETList[minIndex].isFinished)
    ETList.remove(minIndex);
  return min;

function getEvalTree(QueryTreePattern QTP,
QTPNode root, MatchPattern MP, Document Doc)
  if (root.getParent() != null)
    parent = root.getParent();
    upperNodes.add(parent, MP(parent).level);
  end if;
  if (!root.hasChildren)
    strm = stream(MP, root, Doc);
    eNode = new EvalTreeNode(root, upperNodes);
    eNode.setDirectInput(strm);
  else if (root.children.size() = 1)
    upperNodes.add(root, MP(root).level);
    node = root.children[0];
    while (node has only one child)
      upperNodes.add(node, MP(node).level);
      node = node.children[0];
    end while;
    eNode = new EvalTreeNode(node, upperNodes);
    if (node has more than one child)
      for each cNode as child of node do
        cTree = getEvalTree(QTP, cNode, MP,Doc);
        eNode.addChild(cTree);
      end for;
    else
      strm = stream(MP, node, Doc);
      eNode = new EvalTreeNode(root, upperNodes);
      eNode.setDirectInput(strm);
    endif;
  else
    eNode = new EvalTreeNode(root, upperNodes);
    for each cNode as child of root do
      cTree = getEvalTree(QTP, cNode, MP,Doc);
      eNode.addChild(cTree);
    end for;
  endif;
  return eNode;
```

Figure 21. Pseudo code of *S^3.v3* algorithm.

```
class EvalTreeNode-v3 implements NodeInput
constructor(QTPNode mainNode, upperNodes as list
of QTPNodes)
  this.mainNode = mainNode;
  this.upperNodes = upperNodes;

procedure setDirectInput(strm as NodeStream)
  this.directInput = strm;

procedure skip(c as EvalTreeNode-v3)
  key = c.head.getID(this.mainNode);
  while (!c.isFinished &
  c.head.getID(this.mainNode) = key)
    c.next();
  end while;

function advance(c as EvalTreeNode-v3)
  key = c.head.getID(this.mainNode);
  let lc be as an empty array;
  while (!c.isFinished &
  c.head.getID(this.mainNode) = key)
    lc.add(c.head);
    c.next();
  end while:
procedure completeToUpperJoinPoint(res as Match)
  mID = res.getID(this.mainNode);
  for each upNode in upperNodes
    let upID be the id corresponds to upNode based
  on mID;
    res.addID(upNode, upID);

procedure addChild(child as EvalTreeNode-v3)
  this.children.add(child);

procedure open()
  for each $c_i$ in this.children
    $c_i$.open();
  if(this.directInput != null)
    this.directInput.open();
  this.next();
  if(this.head = null)
    this.isFinished = true;
```

Figure 22. Pseudo code of *Evaluation Tree* node used in *S^3.v3.*

```
class EvalTreeNode-v3
procedure next()
 if(!this.outputQueue.isEmpty())
   this.head = this.outputQueue.poll();
   return;
 endif;
 if (this node has no child)
   this.directInput.next();
   res = directInput.head;
   if(res != null)
     this.head = completeToUpperJoinPoint(res);
     return;
   else
     this.head = null;
     this.isFinished = true;
     return;
   endif;
 endif;
 if(OP(this.mainNode) = AND)
   processAND();
 elseif(OP(this.mainNode) = OR)
   processOR();
 if (!this.outputQueue.isEmpty())
   this.head = this.outputQueue.poll();
 elseif(!this.isFinished)
   next();

procedure ANDBalance(inputs as array of
EvalTreeNode-v3)
 let cn this.children.size();
 if(cn > 1)
   for(i = 0; i < cn; )
     l = inputs[i].head.getID(this.mainNode)
     r = inputs[i+1].head.getID(this.mainNode)
     c = compare(l, r)
     while(c != 0)
       if(c > 0)
         skip(inputs[i]);
         if (inputs[i].isFinished)
           return false;
       else if (c < 0)
         i = 0;
         for (j=0; j<i; ++j)
           skip(inputs[j]);
           if (inputs[j].isFinished)
             return false;
       else
         i++;
       endif;
     end while;
   end for;
 end if;
 return true;

 procedure processAND()
 for each $c_i$ in this.children
   if($c_i$.isFinished)
     this.head = null;
     $c_i$.isFinished = true;
     return;
   end if;
 end for;
 b = ANDBalance(this.children);
 if (!b)
   this.head = null;
   $c_i$.isFinished = true;
   return;
 end if;
 for each $c_i$ in this.children
   $lc_i$ = advance($c_i$);
 ANDJnRes = xproduct(lc);
 for each res in ANDJnRes
   res = completeToUpperJoinPoint(res);
   this.outputQueue.offer(res);

function xproduct(lists as array of Match lists)
 if(lists.size = 0) return null;
 let result be an empty list
 for(i=0; i < lists[0].size(); ++i)
   result.add(lists[0][i]);
 for(i=0; i < lists.size(); ++i)
   let newRes be an empty list
   for(j=0; j< result.szie(); ++j)
     for(k=0; k <lists[i].size(); ++k)
       newRes.add(
 result[j].concat(lists[i][k]));
     end for;
   end for;
   result = newRes;
 end for;
 return result;

function ORBalance(inputs as array of
EvalTreeNode-v3)
 sort members of inputs w.r.t. inputs[i].head
 c = -1;
 let selected be an empty list
 for(i = 0; i < inputs.size(); )
   l = inputs[i].head.getID(this.mainNode)
   r = inputs[i+1].head.getID(this.mainNode)
   c = compare(l, r)
   selected.add(inputs[i]);
   if (c < 0)
     break;
   else
     i++;
   endif;
 end for;
 if(c = 0 || inputs.size = 1)
   last = inputs.size() - 1;
   selected.add(inputs[last]);
 endif;
 return selected;

 procedure processOR()
 for each $c_i$ in this.children
   if($c_i$.isFinished)
     this.children.remove($c_i$);
   end if;
 end for;
 if(this.children.size() = 0)
   this.head = null;
   this.isFinished = true;
   return;
 end if;
 selected = ORBalance(this.children);
 for each $c_i$ in selected
   $lc_i$ = advance($c_i$);
 ORJnRes = xproduct(lc);
 for each res in ORJnRes
   res = completeToUpperJoinPoint(res);
   this.outputQueue.offer(res);
 end for;
```

Figure 23. Pseudo code of *Evaluation Tree* node used in S^3.*v3*. (cont'd)

```
class S3-v4 extends S3-v3
 1: procedure execute()
 2:   let parsedQTPs be list of AND-QTPs resulted by parsing Q;
 3:   let SMPList be an empty list
 4:   for each $PQ_i$ in parsedQTPs do
 5:     let $SMP_i$ be execution result of $PQ_i$ against the structural summary of Doc;
 6:     for each $MP_j$ in $SMP_i$ do
 7:       SMPLits.add($MP_j$);
 8:   end for;
 9:   build a DAG using SMPList members, SMPList[i] precedes SMPList[j], if SMPList[i] is subset of SMPList[j];
10:   let superPlans be an array of those members of above DAG that don't have any successor;
11:   for each sp in superPlans do
12:     ET = getEvalTree(Q, Q.root, sp, Doc);
13:     this.ETList.add(ET);
14:   end for;
15:   for each $ET_i$ in ETList do
16:     $ET_i$.open();
17:     if $ET_i$.isFinished
18:       ETList.remove($ET_i$);
19:   end for;
20:   while (true) do
21:     min = nextMatch();
22:     if (min = null)
23:       break;
24:     else
25:       output min;
26:   end while;
```

Figure 24. Pseudo code of S^3.*v4* algorithm.

```
class S3-v5 extends S3-v4
 1: function getEvalTree(QueryTreePattern QTP, QTPNode root, MatchPattern MP, Document Doc)
 2: if (root.getParent() != null)
 3:   parent = root.getParent();
 4:   if (MP(parent) != null)
 5:     upperNodes.add(parent, MP(parent).level);
 6: end if
 7: if (!root.hasChildren)
 8:   strm = stream(MP, root, Doc);
 9:   eNode = new EvalTreeNode(root, upperNodes);
10:   eNode.setDirectInput(strm);
11: else if (root.children.size() = 1)
12:   if (MP(root) != null)
13:     upperNodes.add(root, MP(root).level);
14:   if (root.children[0].hasNOTAxis)
15:     strm = stream(MP, root, Doc);
16:     eNode = new EvalTreeNode(node, upperNodes);
17:     eNode.setDirectInput(strm);
18:     cTree = getEvalTree(QTP, root.children[0], MP, Doc);
19:     eNode.addChild(cTree, true);
20:   else
21:     node = root.children[0];
22:     crn = node.children;
23:     while (crn.size() = 1 & ! crn[0].hasNOTAxis)
24:       if (MP(root) != null)
25:         upperNodes.add(root, MP(root).level);
26:       node = node.children[0];
27:       crn = node.children;
28:     end while;
29:     if (crn.size() = 1 & crn[0].hasNOTAxis)
30:       strm = stream(MP, node, Doc);
31:       eNode = new EvalTreeNode(node, upperNodes);
32:       eNode.setDirectInput(strm);
33:       cTree = getEvalTree(QTP, crn[0], MP, Doc);
34:       eNode.addChild(cTree, true);
35:     else if (crn.size() > 1)
36:       eNode = new EvalTreeNode(node, upperNodes);
37:       hasPOSChid = false;
38:       hasNOTChid = false;
39:       for each child in crn do
40:         cTree = getEvalTree(QTP, child, MP,Doc);
41:         notAxis = child.hasNOTAxis;
42:         if (notAxis)
43:           hasNOTAxis = true;
44:         else
45:           hasPOSChild = true;
46:         eNode.addChild(cTree, notAxis);
47:       end for;
48:       if (!hasPOSChild !! (hasNOTChild && OP(node) = OR))
49:         strm = stream(MP, node, Doc);
50:         eNode.setDirectInput(strm);
51:       endif;
52:     else //node is QTP leaf
53:       if (node.hasNOTAxis & node.parent.children.size() = 1)
54:         eNode = new EvalTreeNode(node.parent, upperNodes);
55:         strm = stream(MP, node.parent, Doc);
56:         eNode.setDirectInput(strm);
57:         cTree = getEvalTree(QTP, node, MP, Doc);
58:         eNode.addChild(cTree, true);
59:       else
60:         eNode = new EvalTreeNode(node, upperNodes);
61:         strm = stream(MP, node, Doc);
62:         eNode.setDirectInput(strm);
63:       endif;
64:     endif;
65:   endif;
66: else // if (root.children.size() > 1)
67:   the same code as lines 36-50
68: endif;
69: return eNode;
```

Figure 25. Pseudo code of S^3.*v5* algorithm.

```
class EvalTreeNode-v5 extends EvalTreeNode-v3
procedure open()
 if (POSChildren.size() = 1 & NOTChildren.size()
 =1 & POSChildren[0] is instance of DirectInput)
   this.isLeaf = true;
   this.directInput.open();
   if (this.directInput.isFinished)
     this.head = null;
     this.isFinished = true;
   else
     res = this.directInput.head;
     this.head = completeToUpperJoinPoint(res);
   endif;
   return;
 endif;
 this.checkForNOT = true;
 for each $c_i$ in this.POSChildren
   $c_i$.open();
 for each $c_i$ in this.NOTChildren
   $c_i$.open();
 this.NOTChildrenNo = this.NOTChildren.size();
 this.next();
 if(this.head = null)
   this.isFinished = true;

procedure addChild(child as EvalTreeNode-v5,
NOTChild as boolean)
 if (NOTChild)
   this.NOTChildren.add(child);
 else
   this.POSChildren.add(child);
 this.children.add(child);

procedure setDirectInput(strm as NodeStream)
 this.directInput = strm;
 this.addChild(new DirectInput(this.directInput),
 false);

procedure next()
 if (this.isLeaf)
   this.directInput.next();
   if (this.directInput.isFinished)
     this.head = null;
     this.isFinished = true;
   else
     res = this.directInput.head;
     this.head = completeToUpperJoinPoint(res);
   endif;
   return;
 endif;
 if(!this.outputQueue.isEmpty())
   this.head = this.outputQueue.poll();
   return;
 endif;
 if(OP(this.mainNode) = AND)
   processAND();
 elseif(OP(this.mainNode) = OR)
   processOR();
 if (!this.outputQueue.isEmpty())
   this.head = this.outputQueue.poll();
 elseif(!this.isFinished)
   next();
procedure processAND()
 for each $c_i$ in this.POSChildren
   if($c_i$.isFinished)
     this.head = null;
     $c_i$.isFinished = true;
     return;
   end if;
 end for;
 b = ANDBalance(this.POSChildren);
 if (!b)
   this.head = null;
   $c_i$.isFinished = true;
   return;
 end if;
 for each $c_i$ in this.children
   $lc_i$ = advance($c_i$);
 ANDJnRes = xproduct(lc);
 Excluded = false;
 for each $c_i$ in this.NOTChildren
   if($c_i$.isFinished)
     this.NOTChildren.remove($c_i$);
     this.children.remove($c_i$);
   end if;
 end for;
 if (this.NOTChildren.size() > 0)
   excluded = this.ANDExclude(ANDJnRes[0]);
 if (!excluded)
   for each res in ANDJnRes
     res = completeToUpperJoinPoint(res);
     this.outputQueue.offer(res);

function ANDExclude(Match res)
 if(this.NOTChildren.size() = 0)
   return false;
 selected = ORBalance(this.NOTChildren);
 rID = res.getID(this.mainNode);
 sID = selected[0].head.getID(this.mainNode);
 c = compare(rID, sID);
 while(c > 0)
   for each $c_i$ in selected do
     this.skip($c_i$);
     if($c_i$.isFinished)
       this.NOTChildren.remove($c_i$);
       this.children.remove($c_i$);
     endif;
   end for;
   if(this.NOTChildren.size() = 0)
     return false;
   selected = ORBalance(this.NOTChildren);
   rID = res.getID(this.mainNode);
   sID = selected[0].head.getID(this.mainNode);
   c = compare(rID, sID);
 end while;
 if(c = 0)
   return true;
 return false;
```

Figure 26. Pseudo code of *Evaluation Tree* node used in *S^3.v5*.

```
class EvalTreeNode-v5 extends EvalTreeNode-v3
procedure processOR()
 for each $c_i$ in this.POSChildren
   if($c_i$.isFinished)
     this.POSChildren.remove($c_i$);
     this.children.remove($c_i$);
   end if;
 end for;
 if(this.children.size() = 0)
   this.head = null;
   this.isFinished = true;
   return;
 end if;
 for each $c_i$ in this.NOTChildren
   if($c_i$.isFinished)
     this.checkForNot = false;
     this.NOTChildren.remove($c_i$);
     this.children.remove($c_i$);
   end if;
 end for;
 selected = ORBalance(this.children);
 for each $c_i$ in selected
   $lc_i$ = advance($c_i$);
 ORJnRes = xproduct(lc);
 if(this.checkForNOT)
   for each res in ORJnRes do
     AllNegative = true;
     for each $c_i$ in this.POSChildren
       node = $c_i$.mainNode;
       if(node != this.mainNode & res.getID(node)
 != null)
         AllNegative = false;
         break;
       end if;
     end for;
     if(AllNegative)
       for each $c_i$ in this.NOTChildren
         node = $c_i$.mainNode;
         if(res.getID(node) = null)
           AllNegative = false;
           break;
         end if;
       end for;
     endif;
     if(AllNegative)
       ORJnRes.remove(res);
   end for;
 endif;
 for each res in ORJnRes
   res = completeToUpperJoinPoint(res);
   this.outputQueue.offer(res);
 end for;
```

Figure 27. Pseudo code of *Evaluation Tree* node used in S^3*.v5*. (cont'd)

```
class S3-v6 extends S3-v5
function groupSuperPattern(superPattern array of
MatchPattern)
 let lf be an array of this.Q's leaves, in
 addition to NOT-Points which have direct input
 sort lf ascending based on distinct number of
 CIDs related to each lf member using
 superPatren.
 group those members of superPattern that are
 equal w.r.t. Output(this.Q) and form NGP as a
 list of GMPs.
 remove super patterns that are participated in
 NGP from superPattern.
 c = 0;
 while superPattern or NGP has ungrouped member
 do
   group Super-Patterns having same CID related
 to lf[c] into GMP[c]
   c++;
 end while;
 let SGMP be array of GMP[i], 0 < i < lf.size()
 return SGMP;

procedure execute()
 let parsedQTPs be list of AND-QTPs resulted by
 parsing Q;
 let SMPList be an empty list
 for each $PQ_i$ in parsedQTPs do
   let $SMP_i$ be execution result of $PQ_i$ against the
 structural summary of Doc;
   for each $MP_j$ in $SMP_i$ do
     SMPLits.add($MP_j$);
 end for;
 build a DAG using SMPList members, SMPList[i]
 precedes SMPList[j], if SMPList[i] is subset of
 SMPList[j];
 let superPatterns be an array of those members
 of above DAG that don't have any successor;

 SGMP = groupSuperPatterns(superPatterns);
 for each GMP in SGMP do
   ET = getEvalTree(Q, Q.root, GMP, Doc);
   this.ETList.add(ET);
 end for;
 for each $ET_i$ in ETList do
   $ET_i$.open();
   if $ET_i$.isFinished
     ETList.remove($ET_i$);
 end for;
 while (true) do
   min = nextMatch();
   if (min = null)
     break;
   else
     output min;
 end while;
```

Figure 28. Pseudo code of S^3*.v6* algorithm.

```
class S3-v6 extends S3-v5
 1: function getEvalTree(QueryTreePattern QTP,
    QTPNode root, GroupedMatchPattern GMP, Document
    Doc)
 2:  if (root.getParent() != null)
 3:    parent = root.getParent();
 4:    upperNodes.add(parent, MP(parent).levels);
 5:  end if
 6:  if (!root.hasChildren)
 7:    strm = groupedStream(GMP, root, Doc);
 8:    eNode = new EvalTreeNode(root, upperNodes,
     GMP);
 9:    eNode.setDirectInput(strm);
10:  else if (root.children.size() = 1)
11:    upperNodes.add(root, GMP(root).levels);
12:    if (root.children[0].hasNOTAxis)
13:      strm = groupedStream(GMP, root, Doc);
14:      eNode = new EvalTreeNode(node, upperNodes,
     GMP);
15:      eNode.setDirectInput(strm);
16:      cTree = getEvalTree(QTP, root.children[0],
     MP, Doc);
17:      eNode.addChild(cTree, true);
18:    else
19:      node = root.children[0];
20:      crn = node.children;
21:      while (crn.size() = 1 & ! crn[0].hasNOTAxis)
22:        upperNodes.add(root, GMP(root).levels);
23:        node = node.children[0];
24:        crn = node.children;
25:      end while;
26:      if (crn.size() = 1 & crn[0].hasNOTAxis)
27:        strm = groupedStream(GMP, node, Doc);
28:        eNode = new EvalTreeNode(node, upperNodes,
     GMP);
29:        eNode.setDirectInput(strm);
30:        cTree = getEvalTree(QTP, crn[0], GMP,
     Doc);
31:        eNode.addChild(cTree, true);
32:      else if (crn.size() > 1)
33:        eNode = new EvalTreeNode(node,
     upperNodes);
34:        hasPOSChid = false;
35:        hasNOTChid = false;
36:        for each child in crn do
37:          cTree = getEvalTree(QTP, child,
     GMP,Doc);
38:          notAxis = child.hasNOTAxis;
39:          if (notAxis)
40:            hasNOTAxis = true;
41:          else
42:            hasPOSChild = true;
43:          eNode.addChild(cTree, notAxis);
44:        end for;
45:        if (!hasPOSChild !! (hasNOTChild &&
     OP(node) = OR))
46:          strm = groupedStream(GMP, node, Doc);
47:          eNode.setDirectInput(strm);
48:        endif;
49:      else //node is QTP leaf
50:        if (node.hasNOTAxis &
     node.parent.children.size() = 1)
51:          eNode = new EvalTreeNode(node.parent,
     upperNodes, GMP);
52:          strm = groupedStream(GMP, node.parent,
     Doc);
53:          eNode.setDirectInput(strm);
54:          cTree = getEvalTree(QTP, node, GMP,
     Doc);
55:          eNode.addChild(cTree, true);
56:        else
57:          eNode = new EvalTreeNode(node,
     upperNodes, GMP);
58:          strm = groupedStream(GMP, node, Doc);
59:          eNode.setDirectInput(strm);
60:        endif;
61:      endif;
62:    endif;
63:  else
64:    the same code as lines 33-47
65:  endif;
66:  return eNode;
```

Figure 29. Pseudo code of *S*^{3}*.v6* algorithm. (cont'd)

```
class EvalTreeNode-v6 extends EvalTreeNode-v5
 1: constructor(QTPNode mainNode, upperNodes as list
    of QTPNodes, GMP as GMP)
 2:  this.mainNode = mainNode;
 3:  this.upperNodes = upperNodes;
 4:  this.GMP = GMP;
 5:
 6: procedure processAND()
 7:  the same code as lines 60-84 in procedure
     processAND of EvalTreeNode-v5 (Figure 26);
 8:
 9:  if (!excluded)
10:    for each res in ANDJnRes
11:      resArr = completeToUpperJoinPoint(res);
12:      this.outputQueue.offerAll(resArr);
13:    end for;
14:
15: procedure processOR()
16:  the same code as lines 2-46 in procedure
     processOR of EvalTreeNode-v5 (Figure 27);
17:
18:  for each res in ORJnRes
19:    resArr = completeToUpperJoinPoint(res);
20:    this.outputQueue.offerAll(resArr);
21:  end for;
22: function completeToUpperJoinPoint(res as Match)
23:  let resArr be an empty list of Matches;
24:  for each MP in this.GMP do
25:    if(res match MP)
26:      newRes = res.clone();
27:      mID = res.getID(this.mainNode);
28:      for each upNode in upperNodes
29:        let upID be the id corresponds to upNode
     based on mID;
30:        newRes.addID(upNode, upID);
31:      end for;
32:      if(!resArr.contains(newRes))
33:        Add newRes to resArr and keep resArr
     sorted ascending base on upperNodes;
34:    endif;
35:  end for;
36:  return resArr;
```

Figure 30. Pseudo code of *Evaluation Tree* node used in *S*^{3}*.v6*.

Appendix B

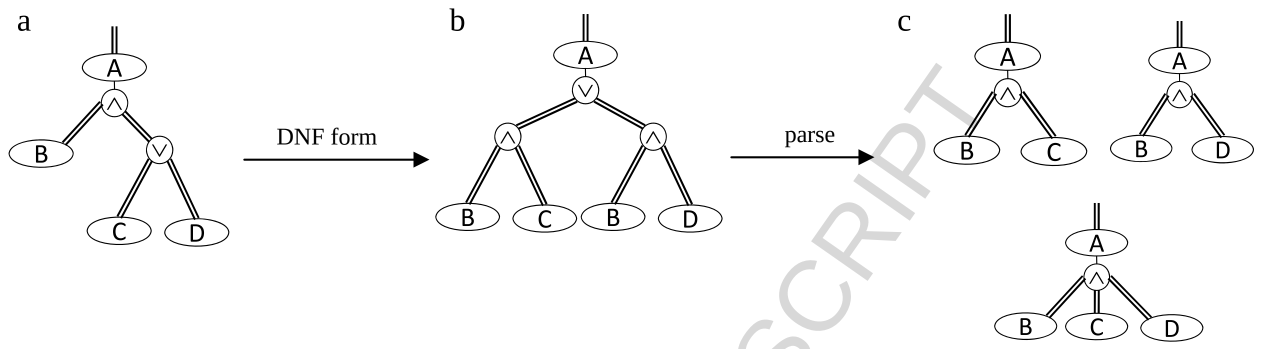


Figure 31. (a) QTP Q_{11}; (b) DNF for of Q_{11}; (c) parsed QTPs

As noted in the paper, our focus is on QTPs with nodes having children connected by *only one* of the *AND* or *OR* logical operators (besides NOT operators). If a QTP has nested logical operators, then its children form a logical expression. Obviously, each logical expression containing nested AND/OR can be rewritten into a Disjunctive Normal Form (DNF), which is a disjunction of conjunctive clauses. Then, each clause can be considered a single child and parsing of QTP is done as described above. For example, consider the QTP Q_{11} in Figure 31. Children of node *A* form the logical expression: $B \wedge (C \vee D)$, which can be transformed into its DNS form: $(B \wedge C) \vee (B \wedge D)$. Therefore, the QTP can be reconsidered as the QTP in Figure 31(b). This new QTP is parsed by considering a conjunctive clause as a single child and using the following transformation: $(B \wedge C) \wedge (B \wedge D) \equiv (B \wedge C \wedge D)$. Therefore, QTP Q_{11} is finally parsed into three AND-QTPs, which are shown in Figure 31(c).

The *Evaluation Tree* is constructed based on the original QTP structure. The only difference is that nested logical operators in the QTP are also considered as join points and a separate node is used for them in the *Evaluation Tree*. The condition of the joining process for this kind of nodes is the same as that for the parent of these nodes. It is worth noting that after the translation of a QTP node into the DNS form, QTP parsing in the presence of logical NOT operators is done in the same way as described above.