TwigTable: using semantics in XML twig pattern query processing

Huayu Wu, Tok Wang Ling, Bo Chen, and Liang Xu

School of Computing, National University of Singapore
{wuhuayu, lingtw, chenbo, xuliang}@comp.nus.edu.sg

Abstract. In this paper, we demonstrate how the semantic information, such as value, property, object class and relationship between object classes in XML data impacts XML query processing. We show that the lack of using semantics causes different problems in value management and content search in existing approaches. Motivated on solving these problems, we propose a semantic approach for XML twig pattern query processing. In particular, we design *TwigTable* algorithm to incorporate property and value information into query processing. This information can be correctly discovered in any XML data. In addition, we propose three object-based optimization techniques to *TwigTable*. If more semantics of object classes are known in an XML document, we can process queries more efficiently with these semantic optimizations. Last, we show the benefits of our approach by a comprehensive experimental study.[1]

1 Introduction

XML query processing has been studied for over a decade. In most XML query languages, e.g., XPath [4] and XQuery [5], queries are expressed as *twig* patterns. Finding all occurrences of a twig pattern query in an XML document is considered the core operation for XML query processing. This process is also referred as *twig pattern query processing* or *twig pattern matching*. More background on XML queries and twig pattern matching is discussed in Section 2.1.

Normally an XML query is composed of structural search and content search. Consider an XPath query Q1 that finds the subject name of the book with the title of "Network", issued to the XML data in Fig. 1:

Q1: //subject[//book/title="Network"]/name

In this query, *//subject[//book/title]/name* is a structural search, aiming to find all matches in the document that satisfy this structural constraint; while the predicate *title="Network"* is a content search, which filters the structural search

[1] This is an extended and updated version of the previously published paper [30]. The major extension includes a discussion on semantics in XML, a technique to process queries across multiple twig patterns, discussions on potential problems of optimized tables, a new optimization scheme with semantics of relationship, and a comparison with the schema-aware relational approach in experiments.

result based on the specified value comparison. Most state-of-the-art XML query processing approaches only focus on employing effective index, e.g., inverted lists (details shown in Section 2.2), to improve the efficiency of performing joins between nodes in more complex twig pattern queries, without distinguishing between structural search and content search. This attempt is proven efficient for structural search without considering values. However, due to the different characteristics between leaf value nodes and internal non-value nodes in XML data, using inverted lists to manage values and to process content search in the same way as structural search will cause problems in managing tremendous number of inverted lists and performing costly structural join for content search.

Besides the inefficiency in content search, which is caused by ignoring the semantic information of value and non-value nodes, existing approaches may also suffer from efficiency problems in structural search. To look deeper into the semantics of non-value document nodes, we can find that a non-value document node may further correspond to an object or a property. Most real life queries aim to find desired objects based on the value predicates on their properties. However, none of existing approaches takes semantics of object and property into account when they manage inverted list index and process queries. The ignorance of such semantics would result in scanning many useless labels in inverted lists during structural search. The details are discussed in Section 3.2 and Section 5.1.

In this paper, we propose a semantic approach for twig pattern query processing. Motivated by solving the problems caused by a lack of using semantics on object, property and value in existing approaches, we propose semantics-based relational tables incorporated with inverted lists of tags to aid twig pattern query processing. In particular, relational tables are used to store values, while inverted lists are used to index internal document nodes, including property nodes and object nodes, but not values nodes. We design *TwigTable* algorithm to perform content search and structural search separately with the two kinds of indexes in twig pattern matching. Content search is performed by table selection before structural search. Because content search is always a predicate between a property and a value, after performing content search the size of the inverted list of the relevant property node is reduced due to the selectivity of the predicate, and the twig pattern query can be simplified by removing value comparisons. Matching a simplified twig pattern with reduced inverted lists for several query nodes will reduce the complexity of structural search, and thus improve the twig pattern matching performance. Finally, the semantic tables can help to extract actual values to answer the queries that ask for property values or object details, which is not efficient to achieve in other structural join based algorithms.

We also need to highlight that the relational tables are constructed based on semantic information such as the relationship among object, property and value. The semantics of property is common for any XML document, i.e., the parent node of each value must be the property of that value. Based on this default semantic information, we initially store each value with its property in the corresponding property table. When more of an object's semantics is known,

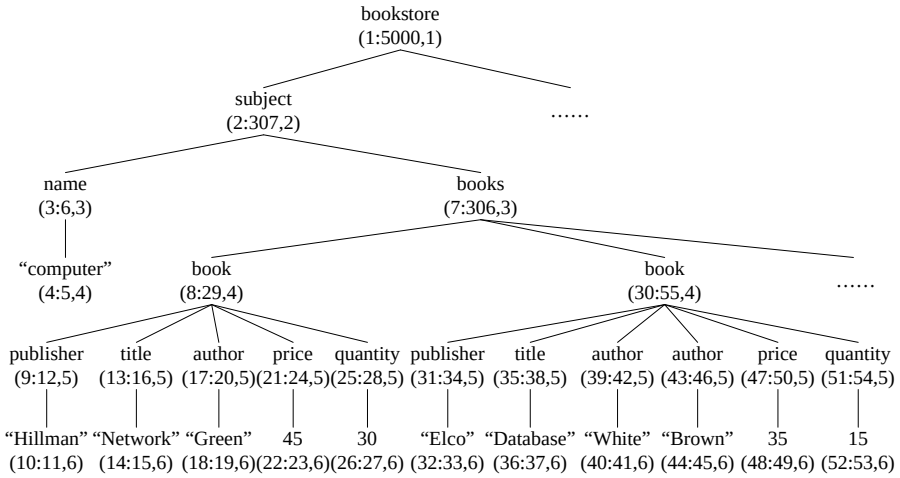


Fig. 1. The bookstore document with all nodes labeled by containment scheme

we propose three optimization techniques to change the tables to be object based. We will show that using object-based tables, a query can be processed even more efficiently. In a word, with more semantic information known, our approach is more efficient to process twig pattern queries.

The rest of the paper is organized as follows. We first describe some background information in Section 2. After that we revisit related work and discuss the motivation of our research in Section 3. The *TwigTable* algorithm with three semantic optimizations is presented in Section 4 and Section 5. We present the experimental results in Section 6 and conclude our work in Section 7.

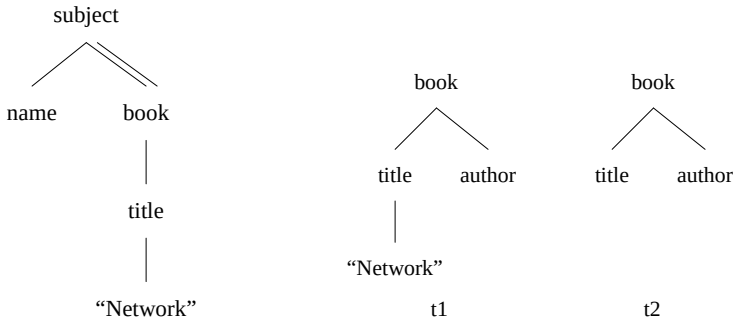
2 Background

2.1 Data model and twig pattern query

Normally an XML document is modeled as an ordered tree, without considering ID references. Fig. 1 (ignoring node labels) shows the tree structure of a bookstore document. In an XML tree, the internal nodes represent the elements and attributes in the document, and the leaf nodes represent the data values which are either a text in an element or an attribute value. Thus a node name is a tag label, an attribute name or a value. Edges in an XML tree reflect element-subelement, element-attribute, element-value, and attribute-value pairs. Two nodes connected by a tree edge are in parent-child (PC) relationship, and the two nodes on the same path are in ancestor-descendant (AD) relationship.

The core query pattern in most standard XML query languages (e.g., XPath and XQuery) is also in a tree-like structure, which is often referred as a *twig pattern*. In particular, an XPath query is normally modeled as a twig pattern query, and an XQuery query is normally modeled as several twig patterns

linked by joins. For example, the XPath query Q1 in Section 1, i.e., *//subject[//book/title="Network"]/name*, can be represented as a twig pattern query in Fig. 2(a). Example 1 shows the twig patterns for an XQuery expression.



(a) Twig pattern for XPath query Q1

(b) XQuery query Q2 can be expressed by joining the two twig patterns t1 and t2 based on author value

Fig. 2. Example twig pattern expressions

Example 1. A query to find the titles of all books written by the author of the book "Network" can be expressed by XQuery:

Q2: For $a in distinct-values(doc("bookstore.xml")
//book[title="Network"]/author)
For $b in doc("bookstore.xml")//book
Where $b/author=$a
Return <book>$b/title</book>

A typical XQuery processor normally models a query by twig patterns to further process it. The XQuery expression Q2 can be transformed into two twig patterns t1 and t2, which are linked by a join on the *author* nodes, as shown in Fig. 2(b).

In a twig pattern query, an edge can be either single-lined or double-lined, which constraints the two matched nodes in either a PC relationship or an AD relationship. Since a twig pattern normally models an XPath expression, we allow the leaf nodes of a twig pattern query to also be a range value comparison or even a conjunction/disjunction of several value comparisons, if the corresponding XPath expression contains such predicates. For example, the twig pattern representation of the XPath query *//book[price>20 and price<30]/title* contains a conjunction of value comparison ">20 and <30" under the query node price.

The process to find all the occurrences of a twig pattern in an XML document is called *twig pattern matching*. A *match* of a twig pattern Q in a document tree T is identified by a mapping from the query nodes in Q to the document nodes in T, such that: (i) each query node either has the same string name as or is evaluated true based on the corresponding document node, depending on whether the query node is an element/attribute node or a value comparison;

(ii) the relationship between the query nodes at the ends of each "/" or "//" edge in Q is satisfied by the relationship between the corresponding document nodes. Matching Q to T returns a list of n-ary tuples to answer Q, where n is the number of nodes in Q and each tuple (a_1, a_2,..., a_n,) consists of the document nodes a_1, a_2,..., a_n, which identify a distinct match of Q in T.

2.2 Document labeling and inverted list

Checking whether two document nodes satisfy the PC or AD relationship specified in a twig pattern pattern query is essential to twig pattern query processing. To facilitate the PC and AD relationship checking, we normally assign a positional label (*label* for short, if no confusion arises) to each document node. There are multiple labeling schemes proposed for XML data, among which the containment scheme [37] and the prefix scheme [26] are most popular. There are also several dynamic encoding schemes [16][33] to avoid re-labeling for dynamic documents. In this paper, we use the containment labeling scheme for illustration, as shown in Fig. 1. It can be replaced by other schemes as long as they are sufficient for PC and AD relationship determination.

Labels are usually organized by inverted lists. Normally, for each type of document node, there is a corresponding inverted list to store the labels of all nodes of this type in document order. To process a query, only relevant inverted lists are scanned to perform structural join based on the query constraints. Because in most algorithms, an inverted list is scanned in a streaming fashion, it is also referred as a *label stream*, or simply a *stream* in some work.

2.3 Semantics of object, property and value in XML data

Object (or entity) is an important information unit in data management, e.g., the ER approach to design a relational database. In data-centric XML databases, object also plays an important role, as most XML queries ask about information of objects. An object normally has several *properties* to describe it from different aspects. Property is also referred as attribute. To differentiate it from attribute type in document schema (e.g., DTD), we use the term *property* in this paper.

In our work, we use semantics of value, property, object and relationship among objects to improve the performance of XML twig pattern query processing. Usually, such semantic information can be provided by the XML database designer. Next we briefly discuss how to discover the semantic information, in case it is not available from database designers.

Identifying values is trivial. Each non-tag text in an XML document is a value, and it must appear as a leaf node in the corresponding document tree. Furthermore, we can also infer that the parent node of each value node in an XML tree corresponds to the property of that value. For example, in Fig. 1 the "Network" is a value and its parent node *title* is the property of this value. This inference of value and property always holds for any document, regardless of whether more semantics is provided or not. Thus our basic algorithm *TwigTable* constructs relational tables based on property and value.

Semantics on object contributes to our optimizations to further improve query processing performance, as presented in Section 5. However, to discover semantics of object and relationship between objects is not so trivial. Intuitively, we may consider the parent node of each property as the corresponding object. For example, in the document in Fig. 1, the *subject* nodes and the *book* nodes are all objects, because they are parent nodes of properties such as name, publisher, title, etc. However, in some documents, properties do not directly connect to their objects. For example, if the document further groups the properties of a book as shown in the two examples in Fig. 3, the parent node of a property may not be the associated object.

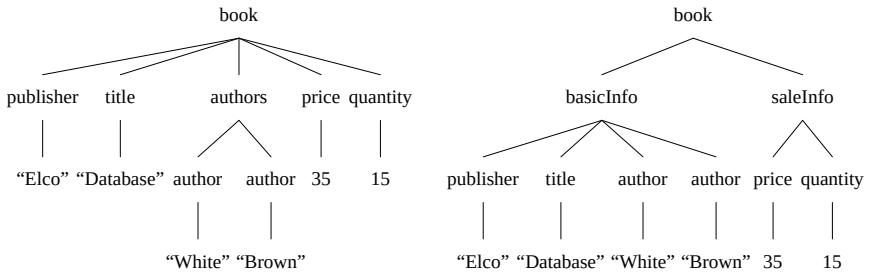


Fig. 3. Two alternative design of book in the bookstore document

Two or more objects that are related to each other are normally reside along the same path in an XML tree. However, the properties of a relationship are not easy to identify. Because of the hierarchical structure of XML data, a property of the relationship between two objects is normally stored under the deeper object of the relationship. This brings difficulty to distinguish it from the properties of that deeper object. Later in Fig. 11 we show an example document in which *quantity* is a property of the relationship between *branch* and *book*, but this property appears as a child node of the deeper object *book*, together with other book properties. The situation may be even more complex when dealing with ternary or n-ary relationships.

There are many attempts to discover the semantics such as object and relationship between objects in an XML document. Generally, there are categorized into three classes.

1. Using available tools and information. There are semantic rich models that work as a schema for XML documents. For example, the ORA-SS [17] model can distinguish between properties, objects and relationships, as well as specify the degree of n-ary relationships and indicate if a property belongs to an object or a relationship. If such model is available alongside an XML document, we can easily discover useful semantic information. Also as semantic web is rapidly developed, there are many ontologies available for different domains. By exploring the ontology of relevant domains, we can have desired semantic information of an XML document.
2. Mining schema or document. Liu et al. [18] infer objects by analyzing DTD. [9] and [35] propose algorithms to discover the semantics such as keys and

functional dependencies in XML documents, which can help to identify objects and relationships. There are also many data mining techniques, such as the decision tree, that can be adopted to infer semantic information.
3. User interaction. In many web-based information management systems [10][25], they use mass collaboration to seek feedback from users to improve semantics identification.

In this paper, we do not offer any new technique to discover objects and relationships in an XML document. All the existing semantics discovery techniques can be used in our work for semantic optimizations. The guideline is that our algorithm is initially built on the semantics of property and value. This information can be correctly inferred from the document structure. When more semantics on object is known, we can incorporate such information into relational table construction, to further improve the performance of query processing.

3 Related work and motivation

3.1 Related work

In the early stage, there are many research efforts on storing and querying XML data using RDBMS. In those relational approaches, they shred XML data into relational tables, and convert XML queries into SQL statements to query the database. The node-based approach [37][13], the edge-based approach [11] and the path-based approach [34][22] shred XML documents based on different kinds of tree components. However, they all suffer from efficiency problems when dealing with structural search. The node-based approach and the edge-based approach need too many costly table joins to process a twig pattern query. The path-based approach is not efficient to handle "//"-axis. The schema-aware decomposition methods [24][6] are proven more efficient than schemaless methods [27], but they are still not efficient for structural search when the document structure is complex. For example, consider a fragment <VP><NP><VP><NP><PP>...</PP></NP></VP></NP></VP> in the TreeBank data [28]. The schema-aware approach can hardly decide what tables between VP and NP should be joined and how many times to join them for the query VP//NP. In short, the relational approach is only suitable for the XML data with regular structure, because of its weakness in structural search.

Later, many native approaches are proposed to process twig pattern queries. One direct native approach is the navigational approach, which traverses tree-structured XML documents to find the occurrences of query patterns. Similar to the navigational approach, some works ([29][23]) transform XML documents and twig pattern queries into sequences, and perform subsequence matching. Both the navigational approach and the subsequence matching approach require a high I/O cost, as the whole document will be considered during query processing.

The structural join based approach is an important class of native approaches to process XML twig pattern queries, and it has attracted most research interest. In early work, Zhang et al. [37] proposed a *multi-predicate merge join* algorithm

based on the containment labeling of an XML document, and showed the superiority over relational approaches. Later an improved stack-based structural join algorithm is proposed by Al-Khalifa et al. [3]. These two algorithms, as well as most of prior works decomposed a twig pattern into a set of binary relationships, i.e., parent-child and ancestor-descendant relationships. Twig pattern matching are done by matching every binary relationship and combining these basic binary matches. The main problem of such approaches is that the intermediate result size may be very large, even when the input and final result sizes are more manageable. To overcome this limitation, Bruno et al. [7] proposed a holistic twig join algorithm, *TwigStack*, which could avoid producing an unnecessarily large intermediate result. However, this algorithm is only optimal for twig pattern queries with only ancestor-descendent relationships. There are many subsequent works [19][15][8][20][14][36] to optimize *TwigStack*, or extend *TwigStack* to solve different kinds of problems. In particular, Lu et al. [19] introduced a *list* structure to make it optimal for queries containing parent-child relationships under non-branching nodes. *TSGeneric* [15] improved the query performance by indexing each inverted list and skipping labels within one list. Chen et al. [8] divided an inverted list into several sub-lists associated to each prefix path or each (tag, level) pair and pruned some sub-lists before evaluating the twig pattern. Lu et al. [20] used Extended Dewey labeling scheme and scanned only the labels of leaf nodes in a twig query. [14] and [36] extended twig pattern query to support OR-predicate and NOT-predicate separately. However, all these structural join based work only focus on structural search. For the value node in each query predicate, they normally treat it the same as element node and perform structural joins for the whole query structure. As a result, they suffer from several problems as mentioned in the next section.

3.2 Motivation

The structural join based approaches are proven efficient in structural search. However, because they do not consider the semantics of value and other types of document nodes, they suffer from several problems during query processing.

1. **Inverted list management.** In most structural join based approaches, all nodes including elements, attributes and values in an XML tree are labeled and the labels of each type of nodes are organized in an inverted lists. When we build inverted lists for values, the number of different values causes a problem of managing a tremendous number of inverted lists. Based on our investigation, a 100MB XML document contains over 300 thousand different values, which correspond to 300 thousand inverted lists. This number will linearly increase according to the document size increase.
2. **Advanced content search.** Since twig pattern query normally models XPath expression, the advanced content search, such as numeric range search, containment search or even conjunction/disjunction of several value comparisons, which often appear in XPath query predicates may also appear as a leaf node in a twig pattern query. Without handling values specially, existing

approaches have difficulty in supporting these advanced content search. For example, to process a query to find the books with the price greater than 15, it is time consuming to get all the inverted lists with the numeric names greater than 15, and combine labels in them by document order, to perform this range search. Also structural join with inverted lists can hardly support containment search, such as //book[contains(@title, 'XML query')]/price.

3. **Redundant search in inverted lists.** Inverted lists for values do not have semantic meanings. This may cause redundant search during inverted list scanning. For example, when a query is interested in books with the price of 35 in the bookstore document, structural search scans the inverted list for the value node '35' (denoted by T_{35}). Since in T_{35} we do not differentiate whether a label corresponds to *price* or *quantity*, we need to check all labels in this inverted list though many of them stand for *quantity* of 35, and definitely do not contribute to the query result.
4. **Actual value extraction.** To answer a query, what we need is not twig pattern occurrences represented as tuples of labels, but value results. For example, after finding a number of occurrences of the twig pattern query in Fig. 2(a), we need to know the value under each *name* node. One major advantage of the structural join based approaches is that they only need to load relevant inverted lists to process a query, instead of scanning the whole document with high I/O cost. However, when a query asks for values of a certain property, after getting a set of resulting labels of that property from pattern matching, they cannot find the child value under each label using inverted lists. To extract actual values, they have to read the document again, which violate the initial objective in I/O saving.

Motivated on solving all these problems, we propose a semantic approach that uses both inverted lists and relational tables to perform twig pattern matching.

4 TwigTable algorithm

4.1 An overview of TwigTable

In *TwigTable*, we pay attention to the semantics of value during index construction. We maintain inverted list index only for structural nodes (internal nodes). For each value node, we store it into a relational table index[2] with the label of its property node, instead of labeling it and putting its label into an inverted list as other approaches do. Then the number of inverted lists is limited to the number of different element/attribute types in the document.

Query processing in *TwigTable* includes three steps. In the first step, we perform content search for the value comparisons in query predicates, using relational tables. After that, the query is rewritten by removing all value comparisons. In the second step, we perform structural search for the simplified query

[2] To avoid the overhead on maintaining relational tables, the relational table index can also be replaced by other types of index to bidirectional map values and their properties. However, the trade-off is the inconvenience to support advanced content search and to meet the requirements in our object-related optimizations.

pattern, using any structural join algorithms, e.g., *TwigStack*. The last step is to return result to users. If the output node is a property type, the value result can be extracted from tables, instead of accessing the original document.

4.2 Document parsing in TwigTable

When we parse an XML document, we only label elements and attributes, and put the labels into corresponding inverted lists. Values in the document are not labeled, instead we put them into relational tables together with the labels of their parent *property* nodes. Normally this parsing step is only executed once for an XML document, and after all relevant indexes are properly built during the parsing step, the system is ready to process twig pattern queries over the given document. The detailed algorithm *Parser* is presented in Algorithm 1.

Algorithm 1 Parser

Input: A *SAX* stream of the given XML document
Output: A set of inverted lists and a set of relational tables

```
1: initialize Stack S
2: while there are more events in SAX stream do
3:     let e = next event
4:     if e is a start tag then
5:         //step 1: label elements
6:         create object o for e
7:         assign label to o
8:         push o onto S
9:         for all attributes attr of e do
10:            //attributes are parsed in the same way as elements.
11:            assign label to attr
12:            put label of attr into the inverted list T_attr
13:            insert the label of attr and the value of attr into the table R_attr
14:        end for
15:        //step 2: put labels of elements into inverted lists
16:        put label of o into the inverted list T_e
17:    else if e is a value then
18:        set e to be the child value of the top object in S
19:    else if e is an end tag then
20:        // step 3: Insert values with their parent element into tables
21:        pop o from S
22:        if o contains a child value then
23:            insert label of o together with its child value into table R_e
24:        end if
25:    end if
26: end while
```

We use the *SAX* to read the input document and transform each tag and value into events. Line 3 captures the next event if there are more events in the SAX stream. Based on different types of events, different operations are performed accordingly. Line 4-16 are executed if the event e is a start tag. In this case, the first two steps are triggered. The system first constructs an object for this element and assigns a label to it. It then puts the label into the inverted list for that tag. A stack S is used to temporarily store the object so that when an end tag is reached, the system can easily tell on which object the operation will be executed. At line 9-14, the system analyzes the attributes for an element if any. Based on the same operating steps, it labels the attributes and puts labels into inverted lists. The attribute values are treated in the same way as element

values. Line 17-18 is the case that the event is a value type. Then the value is simply bound to the top object in S for further table insertion. When the event is an end tag in line 19-25, the last step is performed, which is popping the top object o from S and inserting the label of o together with its value into the relational table for o, if it has a value.

Example 2. After parsing the bookstore document, the new labeled document tree is shown in Fig. 4. Comparing to the document tree in Fig. 1, we can see that *TwigTable* does not label value nodes. The advantages of this attempt are reported in Section 6.

Fig. 4. The bookstore document with only internal nodes labeled, during *TwigTable* parsing

Some example relational tables which store data values are shown in Fig. 5. The name of each table is a property name, and each table contains two fields, the label of the property node and the corresponding child value. We call these tables *property* tables.

$R_{publisher}$

label	value
(7:8,5)	Hillman
(19:20,5)	Elco
…	…

R_{title}

label	value
(9:10,5)	Network
(21:22,5)	Database
…	…

R_{author}

label	value
(11:12,5)	Green
(23:24,5)	White
(25:26,5)	Brown
...	...

Fig. 5. Example property tables

Similar to inverted list, relational table also plays an index role to aid twig pattern query processing. As a result, how to cope with document updates is an important issue for both inverted lists and relational tables. Inverted lists

have been widely used and well studied for years. There are different ways to maintain an inverted list for updates, e.g., employing a B^+ tree on each list. Here we discuss the maintenance of relational tables when the XML document is updated. Actually the relational table is easy to maintain. If the document is dynamic with frequent updates, we can adopt a dynamic labeling scheme to label the document so that the update will not cause re-labeling of remaining document nodes. Thus, any deletion or insertion only brings the update of the relevant tuple, without affecting other tuples.

4.3 Query processing in TwigTable

As mentioned in Section 4.1, query processing in *TwigTable* contains three steps: content search and query rewriting, structural search, and value extraction. Theoretically, the first two steps, i.e., content search and structural search, can be reordered. The reason why we perform content search before structural search is that content search normally results in high selectivity. By performing content search first, we can reduce the complexity of structural joins. This is similar to selection push-ahead in relational query optimizers. The pseudo-code of *TwigTable* query processing is presented in Algorithm 2.

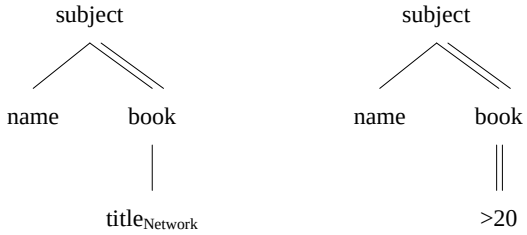
Algorithm 2 TwigTable query processing

Input: A query Q and necessary inverted lists and relational tables
Output: A set of value results answering Q
1: //step 1: perform content search, construct new inverted lists and rewrite the query
2: **while** there are more value comparisons in predicates of Q **do**
3: let c be the next value comparison, and p be its property (parent element or attribute)
4: create a new inverted list $T_{p'}$ for p
5: select the labels based on c from the table R_p, and sort the resulting labels by document order
6: put the selected labels into $T_{p'}$
7: rewrite the query to replace the sub-structure p/c by p'
8: **end while**
9: //step 2: perform structural search on the rewritten query with new inverted lists
10: process the rewritten pattern of Q using any existing efficient structural join algorithm like *TwigStack*, to get labels for output nodes
11: remove newly created inverted lists
12: //step 3: return query answer
13: extract actual values with labels from corresponding tables, if the output node is a property node; otherwise access the document to return subtrees.

We first perform content search in Line 2-8. The algorithm recursively handles all value comparisons in two phases: creating new inverted lists based on the predicates and rewriting the query to remove the processed value comparisons. In more details, Line 3-6 execute *SQL* selection in the corresponding property tables based on each value comparison, and then put all the selected labels, which satisfy the value comparison, into the new inverted list for the corresponding property node. Line 7 rewrites the query in such a way that every value comparison and its parent property are replaced by a new query node which has an identical name as the corresponding new inverted list. The second step is using *TwigStack* or other efficient structural join algorithms to process the simplified query with new inverted lists in Line 10-11. Last in line 13, we return result

based on labels of output nodes. In particular, we can extract actual values from the corresponding table, if the output node is a property node.

Example 3. We use the twig pattern query in Fig. 2(a) to illustrate how *TwigTable* works. In the first step, *TwigTable* identifies the only predicate with value comparison is *title="Network"*. During content search, *TwigTable* executes an *SQL* selection in the table R_{title} to get all the labels of the element *title* which have a value of "Network". Then we put the selected labels into a new inverted list for *title*, $T_{title'}$, and rewrite the twig pattern query to replace the sub-structure of the node *title* and its child node *"Network"* by *title'*, as shown in Fig. 6(a). The new query node *title'* corresponds to the newly created inverted list $T_{title'}$, in which all labels satisfy the constraint *title="Network"*. To clearly explain *title'* in the rewritten query, we use $title_{Network}$ in Fig. 6(a). Finally we use a twig pattern matching algorithm, e.g., *TwigStack* to process the rewritten query in Fig. 6(a), with the inverted list $T_{title'}$ for the node $title_{Network}$.



(a) Rewritten query example (b) Invalid query example

Fig. 6. A rewritten query example and an invalid twig pattern query example

As described in Section 2.1, twig pattern query is an intermediate query representation of formal XML query languages, e.g., XPath and XQuery. Since in the predicate of an XPath or XQuery query, a value must link to a property through an operator, e.g., price>20, the value comparison in a twig pattern query must be a child ('/'), instead of a descendant ("//") of an internal query node. A twig pattern expression shown in Fig. 6(b) is invalid, as the value comparison follows a "//" edge. Semantically, this query cannot be well interpreted; and practically, this query will never appear in XPath or XQuery expressions. We do not consider such invalid twig patterns, thus our algorithm can perform any content search using property tables.

Note that *TwigTable* saves I/O cost in value extraction when the output node is a property node, because we do not need to visit the original document, but only access relevant relational tables to find values. However, if the output node matches some internal nodes with subelements in the document, the result should be the whole subtree rooted at each matched node, instead of a single value. In this case, *TwigTable* has no advantage in result return over other approaches.

4.4 Analysis of TwigTable

Label and inverted list management *TwigTable* combines values to their parent elements, and avoids labeling value nodes separately. Then the num-

ber of labeled nodes in memory will be greatly reduced. Moreover, *TwigTable* puts values into relational tables, instead of maintaining separate inverted lists for them. Thus the problem of managing a tremendous number of inverted lists in previous work can be solved.

Content search *TwigTable* organizes values based on their property semantics in tables. When the value in a query predicate has different semantic meaning, i.e., corresponds to different properties, *TwigTable* only accesses the correct property table to search the value. In contrast, other approaches have to scan all such values to perform structural join, though many of them correspond to other properties and definitely do not contribute to the result.

Inverted list searching reduction Performing content search before structural search in *TwigTable* can significantly reduce the size of relevant inverted lists. Consider the query in Fig. 2(a). Assume there is only one book called "Network". If the number of different books is b, the size of the inverted list for the property *title* is also b in previous approaches. We need $O(b)$ to scan all the labels in the inverted list for *title*. *TwigTable* processes selection in advance, so that the new inverted list for *title* is created based on the value "Network". In this case the new inverted list has only one label inside based on our assumption. Normally, when the selectivity of a property is high, like in this example, *TwigTable* can significantly improve the efficiency of structural search by greatly reducing the inverted list size for this property.

Advanced search support Since *TwigTable* can use any existing RDBMS to manage property tables, all the advanced searches which are supported by the relational system are also supported in *TwigTable*.

We can observe that sequential scans and structural joins for labels of both property node and value node in previous work are replaced by selections in semantic tables in *TwigTable*. Actually in any relational database system, such table selection can be done very efficiently. It is not surprising that replacing structural join by selection for content search will improve the overall performance.

Generally, *TwigTable* gains benefit from performing content search ahead of structural search, and then reduce the complexity of structural search. Thus most advantages discussed in this section hold only for queries with value predicates, which are commonly seen in real life. When a query does not have value comparison as predicate, we just follow any existing structural join algorithm to perform structural search directly.

4.5 Queries across multiple twig patterns

A twig pattern can be used to model a simple query, e.g., a query that can be represented by XPath. When a query is more complex, we need to model it with multiple twig patterns and value-based joins are used to connect these twig patterns. One example is shown in Fig. 2(b). As pointed out by [7], structural join based algorithms can only efficiently process single-patterned queries. When

a complex query involves several twig patterns, either from the same document or across different documents, structural join based algorithms will fail to work.

The reason why structural join based twig pattern matching algorithms cannot process queries involving several twig patterns is that those algorithms cannot perform value-based join between twig patterns using their inverted list indexes. One naive approach is to match different twig patterns in such a query separately. By considering each query node that are involved in value-based join as an output node, they can then access the original document to retrieve the child values for these query nodes. Last, they join the matching results from different twig patterns based on the retrieved values. Obviously this attempt is I/O costly, and also may produce a large size of intermediate result.

In *TwigTable*, we introduce relational tables to store values. This structure offers an opportunity to process queries across multiple twig patterns. We observe that a join operation between two twig patterns is based on a value comparison between two properties in the two twigs. Using property tables, we can easily perform the value based join. We use an example to illustrate how *TwigTable* is extended to process such queries.

Example 4. Consider the query in Fig. 2(b). There are two twig patterns t1 and t2 are involved in this query. First, *TwigTable* estimates the selectivity of each twig pattern. In this case, obviously t1 has a higher selectivity. Then *TwigTable* matches t1 to the document, to get the value result of query node *author*. Due to the high selectivity on *title="Network"* w.r.t. the data in Fig. 4, the matching result only returns one author name, which is "Green". In the next step, *TwigTable* joins the value result from the first twig pattern to the property table which corresponds to the joining node in the second twig pattern. In this example, we join the only value "Green" to R_{author}, to get a list of labels such that all of them correspond to the value "Green". Finally, we form a new inverted list with the selected labels for the *author* node, and match t2 to the document.

Discussion *TwigTable* uses both inverted lists and tables for twig pattern matching, which offers a good opportunity to process queries across pieces of a document or across different documents. However, to process such a query, an optimizer is necessary to decide which twig pattern should be matched first, to reduce the searching space for other twig patterns. Such an optimizer is quite similar to a relational optimizer, and needs to estimate the cost to matching each twig pattern, the selectivity of each twig pattern, and also the cost and the selectivity of each value-based join between twig patterns. In this paper, we show the capability of *TwigTable* to process queries across multiple twig patterns. How to generate an optimal query plan to evaluate such queries will be further investigated.

5 Semantic optimizations

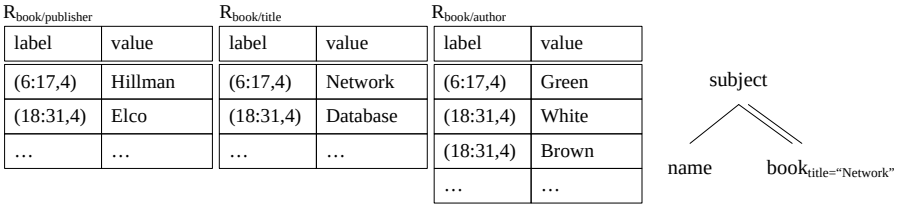
Tables in *TwigTable* are built based on the semantic relationship between property and value. That is why we call them property tables. Using property tables

to perform content search may still not be efficient enough in some cases. Since object is an important information unit for most queries, we can optimize the property tables to be object based, to further improve the performance.

5.1 Optimization 1: object/property table

Motivation: Using property tables may still suffer from redundant search in relevant inverted lists. Consider the query in Fig. 2(a). Supposing there are b books in the bookstore and only one of them is called "Network". After *TwigTable* rewrites the query in Fig. 6, the size of the inverted list for *title* is reduced to 1. However the size of the inverted list for *book* is still b, though we know one label in it matches the label in the *title* inverted list. To solve this efficiency problem, we propose an optimization scheme based on the object semantics.

Optimization: Instead of storing each value with the label of its associated property node, we can put the property value and the label of the corresponding object node into relational tables. For example, in the bookstore document we put values of *publisher*, *title* and so forth with labels of the corresponding object *book* into *object/property* tables as shown in Fig. 7(a). The 'label' field of each table stores the label of the object and the following 'value' corresponds the value of different properties in different tables. When we perform a content search, we can directly select the object labels in the corresponding object/property tables and construct a new inverted list for the object. To process the query in Fig. 2(a), we perform the content search using $R_{book/title}$ to restrict the book labels based on the condition on title value. After that the query can be further rewritten accordingly, as shown in Fig. 7(b), where $T_{book'}$ is the new inverted list for the element *book* and we use $book_{title="Network"}$ to explicitly explain *book'*. Here we not only reduce the size of T_{book}, but also further reduce the number of structural joins and the number of query nodes by one. Then we can get better performance when we execute the simplified query.



$R_{book/publisher}$

label	value
(6:17,4)	Hillman
(18:31,4)	Elco
…	…

$R_{book/title}$

label	value
(6:17,4)	Network
(18:31,4)	Database
…	…

$R_{book/author}$

label	value
(6:17,4)	Green
(18:31,4)	White
(18:31,4)	Brown
…	…

(a) Tables in *TwigTable* Optimization 1

(b) Rewritten query for Fig 2(a)

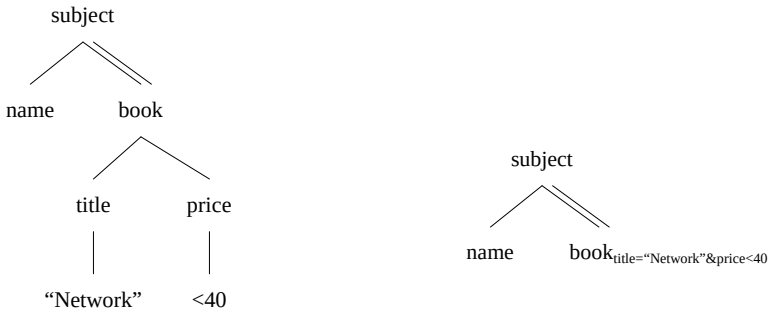
Fig. 7. Tables and rewritten query under *TwigTable* Optimization 1

This object-level optimization is general to all queries with a predicate on a property to constraint an object. For the case that the same property type belongs to multiple object classes, this optimization gains even more benefits, as it avoids accessing those property nodes that do not belong to the required object class by distinguishing their object semantics.

Discussion: [Ordinal Column] This optimization may lose order information for multi-valued properties. Such information may be important in some cases. For example, the order of authors is important, but from the book/author table we cannot tell which author comes first for a certain book. To solve this limitation, we can simply add an additional column in the corresponding object/property table for such a multi-valued property, to indicate the ordinal information.

5.2 Optimization 2: object table

Motivation: It is quite normal that some queries contain multiple predicates on the same object. Consider the query shown in Fig. 8(a), which aims to find the subject of the book with the title of "Network" and the price less than 40. To answer this query, Optimization 1 needs to find the labels of the books whose title is "Network" and the labels of the books whose price is less than 40 separately using the object/property tables, and intersect them. With more semantic information, we know that *title* and *price* are both properties of the object *book*. If we have one table for this object that contains the both properties, books satisfying these two constraints can be found directly with one *SQL* selection.



(a) Query with multiple value predicates under the same object

(b) Rewritten query for Fig 8(a) under Optimization 2

Fig. 8. Example query with multiple value predicates under the same object and its rewritten query in Optimization 2

Optimization: A simple idea is to merge the object/property tables in Optimization 1 for each object class. For multi-valued properties, such as *author* in our example, it is not practical to merge it with other properties. In this case, we can merge all the single-valued properties of an object into one object table and keep the object/property tables for multi-valued properties. The resulting tables for the object class *book* in the bookstore document under this optimization are shown in Fig. 9. In R_{book}, each label of book is stored with all the single-valued property values of that book. When we process queries with multiple predicates on single-valued properties of an object, we can do a selection in that object table based on multiple constraints in one time. For example, to process the query in Fig. 8(a), we can select book labels based on the two predicates in R_{book}

with one SQL selection. Then the original query can be rewritten as shown in Fig. 8(b). Comparing with the Optimization 1 approach, we further simplify the query and prune intermediate results for the two predicates.

R_{book}

label	publisher	title	price	quantity
(6:17,4)	Hillman	Network	45	30
(18:31,4)	Elco	Database	35	15
…	…	…	…	…

$R_{book/author}$

label	value
(6:17,4)	Green
(18:31,4)	White
(18:31,4)	Brown
…	…

Fig. 9. Tables for *book* in the bookstore document under *TwigTable* Optimization 2

Discussion: [Mixed table selection] If the multiple predicates involve both single-valued properties and multi-valued properties, we can intersect the selection result from the object table for the single-valued properties, with the selection result from the object/property tables for the multi-valued properties.

[Rare property] Properties may optionally appear under the associated objects. In some cases, the occurrence of certain properties may be rare. We call such properties *rare properties*. Suppose in the bookstore document, only a few books have a *second_title*, then *second_title* is a rare property. If we put this rare property as a column in the book table, there will be too many NULL entries. Some relational database systems can deal with sparse attributes in the physical storage. In case some other systems do not have this function, we can maintain a separate table specially for all rare properties. The rare property table contains: the object name, the rare property name, the object label and the property value. Suppose in the bookstore document, *second_title* is a rare property of *book* and *sale_region* is a rare property of *magazine*, the example rare property table is shown in Fig. 10. Queries involving rare properties are processed by accessing the rare property table with the object name and the property name.

$R_{rare\_property}$

object	property	label	value
book	second_title	(76:89,4)	An introduction to data mining
magazine	sale_region	(128:143,4)	Singapore
book	second_title	(282:299,4)	A first course
…	…	…	…

Fig. 10. Table for rare properties

[Vertical partitioning] An object table is obtained by merging the object/property tables for all the single-valued property types under the same object class. When there are many single-valued property types under a certain object class, the tuple size of the corresponding object table will be very large, which results in a

high I/O cost in selection. A common way in RDBMS design to reduce such I/O cost is the vertical partitioning of a table ([21]). We can refer to the query history to see which properties often appear together in the same query, and then split the original object table into several partitions according to such information. Since vertical partitioning is not a new technique, we do not discuss it any more.

5.3 Optimization 3: relationship table

Motivation: The hierarchical structure of an XML document cannot reflect the relationships between objects explicitly, however, such relationships do exist usually. Consider another design of the bookstore document as shown in Fig. 11, in which the books are also grouped by different branches. In Fig. 11, the *quantity* is not a property of book, but a property of the relationship between *branch* and *book*. Putting a relationship property into the object table of the property's nearest object does not affect the accuracy of query processing. Consider a query to find the code of the branch that has some computer book with a low quantity, i.e., less than 20, expressed in Fig. 12(a). If we have no idea on the relationship between *branch* and *book*, but store the property *quantity* in the object table for *book*, Optimization 2 will rewrite the query as in Fig. 12(b) for further matching. However, since we aim to find qualified *branches*, matching the *book* node is redundant. If we know the predicate is on the relationship between *branch* and *book*, we may ignore *book* during pattern matching to improve efficiency.

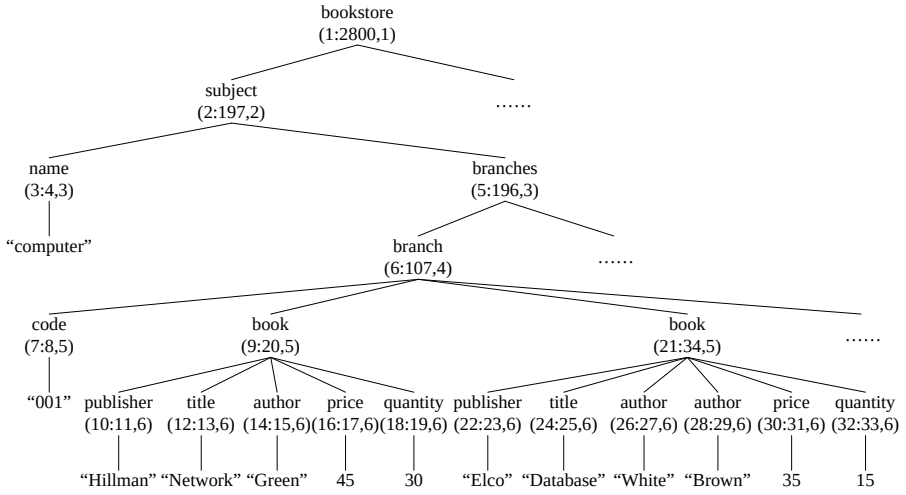
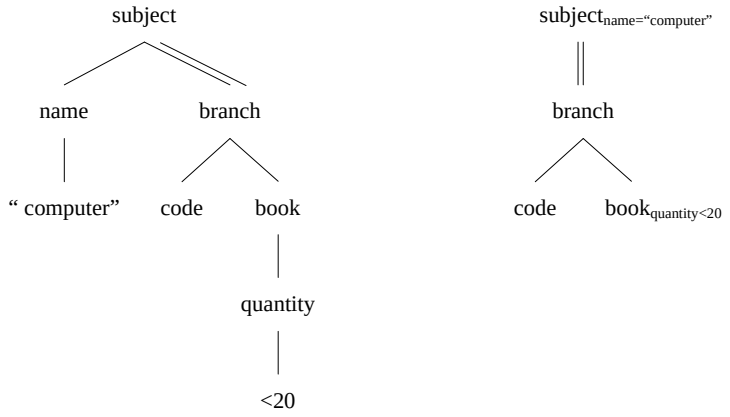


Fig. 11. Another design of the bookstore document

Optimization: If we have the semantic information about the property of relationship, we can introduce relationship tables. A relationship table store the property value and the label of the participating objects of each relationship instance. The example relationship table for the document in Fig. 11 is shown

in Fig. 13(a). When a relationship involves more than two objects, the corresponding relationship table will include the labels of all the objects. Using the relationship table, the query in Fig. 12(a) can be rewritten as in Fig. 13(b). Compared to Optimization 2, the query is further simplified with the semantics of relationship, and the size of the inverted list of the branch node is reduced. Then the query processing performance will be further improved.



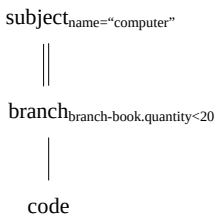
(a) Example twig pattern query (b) Rewritten query for Fig 12(a) under Optimization 2

Fig. 12. Example query with predicate on relationship property and its rewritten query in Optimization 2

$R_{branch\text{-}book}$

$label_{branch}$	$label_{book}$	quantity
(6:107,4)	(9:20,5)	30
(6:107,4)	(21:34,5)	15
…	…	…

(a) Relationship table



(b) Rewritten query for Fig 12(a) under Optimization 3

Fig. 13. Example relationship table and rewritten query in *TwigTable* Optimization 3

Discussion: [Overlapping predicates] A query node can be involved in both an object predicate and a relationship predicate. For example, if the query in Fig. 12(a) has an additional predicate on book price, the query node *book* will be involved in two predicates (i.e., the relationship predicate and the object predicate) overlapping on it. To handle overlapping predicates, we can perform the content search based on different predicates separately, and then intersect the label results to construct the temporary inverted list for the involved object.

[Merging object table and relationship table] If the semantics of participation constraint between two object classes is known, we can merge the object table(s)

and the relationship table when the constraint is many-to-one or one-to-one. This is similar to the translation from ER diagram to tables with the consideration of participation constraints in relational database design. However, similar to the vertical partitioning, how to physically maintain relational tables is generally bound to the performance analysis for practical queries.

5.4 A summary

The optimization techniques are proposed around the semantics of object. This object-level attempt is motivated by the fact that most queries ask about the information of objects. Generally, as more semantic information is known, we can optimize *TwigTable* to different levels, to get better performance.

When the object information is not available but we still need it to manage data, we can roughly treat the parent node of each property as an object. As indicated in Section 2.3, this inference may not be correct in some cases, but it does not affect the correctness of twig pattern query processing. However, we construct table index based on the semantics of object instead of the simple structural information is because that object is the information unit in real life queries. Optimizations in object level can minimize the number of structural joins without affecting the processing of general queries. For example, if a *person* has a composite property *name* with *firstName* and *lastName*. Because most queries are issued to the object *person*, instead of to the property *name* only, if we construct table based on *name*, with the property of *firstName* and *lastName*, after the content search, we still have to join *name* with *person*. However, if we build the table based on the real object *person*, i.e., incorporating *name/firstName* and *name/lastName* into *person* table, any predicate on *firstName* or *lastName* will result in a label filtering for *person* directly, to save a structural join.

6 Experiments

In this section, we conduct experiments to show the advantage of *TwigTable*. We first compare our approach to a schema-aware relational approach [24], which is considered more efficient than other relational approaches. Then we compare *TwigTable* and its two optimizations to *TwigStack*, a typical structural join based twig pattern matching algorithm. Note that in this experiment, our algorithms take *TwigStack* to perform structural search, thus we compare to *TwigStack* to show the benefit gained. We can also take any other structural join algorithm to perform structural search. We do not compare with them because the comparison with *TwigStack* is sufficient to show the advantage of our approach.

6.1 Settings

We implemented all algorithms in Java. The experiments were performed on a dual-core 2.33GHz CPU and a 4GB RAM under Windows XP.

We used three types of real-world and synthetic data sets to compare the performance of *TwigStack* and our approaches: NASA [1], DBLP and XMark

[32]. NASA is a 25MB document with deep and complex schema. DBLP data set is a 127MB fragment of DBLP database. It is rather regular with a simple DTD schema but a large amount of data values. We also used 10 sets of XMark benchmark data with sizes from 11MB to 110MB for our experiments.

We selected three meaningful queries for each data set. All the queries contain predicates with value comparisons, as value predicates appear in most practical queries. Generally, there are three types of query predicates: predicates of equality comparison, predicates of range comparison and multiple predicates of different comparisons under one object. The queries are shown in Fig. 14.

In TwigTable, we use the Sybase SQL Anywhere [2] to manage relational tables, and inherit the default database parameters.

Data Set	Query	Path Expression
NASA	NQ1	//dataset//source//other[date/year>1919 and year<2000]/ author/lastName
	NQ2	//dataset/tableHead[//field/name='rah']//tableLinks //title
	NQ3	//dataset//history//ingest[date[year>1949 and year<2000] [month='Nov'][day>14 and day<21]]//creator/lastName
DBLP	DQ1	/dblp/article[/author='Jim Gray']/title
	DQ2	/dblp/proceedings[year>1979]/isbn
	DQ3	/dblp/inproceedings[title='A Flexible Modeling Approach for Software Reliability Growth'][year='1987'][author='Sergio Bittanti']/booktitle
XMark	XQ1	//regions/africa/item[//mailbox//mail/from='Libero Rive']// keyword
	XQ2	//person[//profile/age>20]/name
	XQ3	//open_auction[//bidder[time>18:00:00]/increase>5]/quantity

Fig. 14. Experimental queries

6.2 Comparison with Schema-aware Relational Approach

In this section, we compare *TwigTable* with a Schema-aware Relational Approach proposed in [24]. We name it *SRA* for short. We also use the Sybase SQL Anywhere for *SRA*. Since this approach is weak in dealing with "//"-axis, we adopt the proposal in [12] to augment it. *SRA* is proven more efficient than other schemaless relational approach to process XML queries. The execution time for both *SRA* and *TwigTable* to process the queries in NASA, DBLP and a 110MB XMark data is shown in Fig. 15. Note that the Y-axis is in logarithmic scale.

From Fig. 15 we can see that for NASA and XMark data (NQ1-3, XQ1-3) *TwigTable* is more efficient, but for DBLP data (DQ1-3) *SRA* is more efficient.

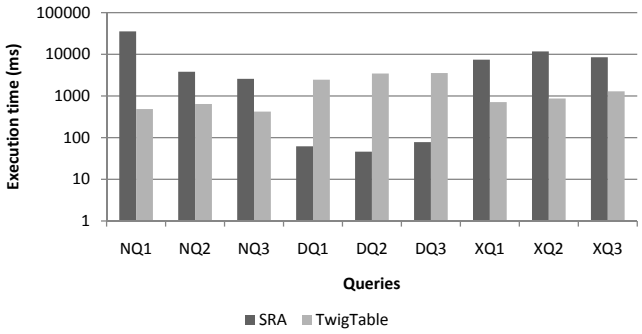


Fig. 15. Comparison result between *SRA* and *TwigTable*

DBLP data is rather regular and flat, thus the values in DBLP data can be perfectly shredded into relational tables. The maximum height of DBLP is 4, which means using *SRA* shredding method, there is at most one table join required for all queries and this join is between the table for root (containing only one tuple) and another table. For such a relational-like document, the relational approach is much more efficient, because table selection dominates the overall performance and this operation can be performed very efficiently in all relational databases.

However, most real life XML data is not as regular as DBLP, otherwise, it violates the advantage of the semi-structured format. As we see for NQ1-3 and XQ1-3, when the document is deeper and more complex, i.e., requires more table joins for *SRA*, the performance of *SRA* is badly affected.

6.3 Comparison with TwigStack

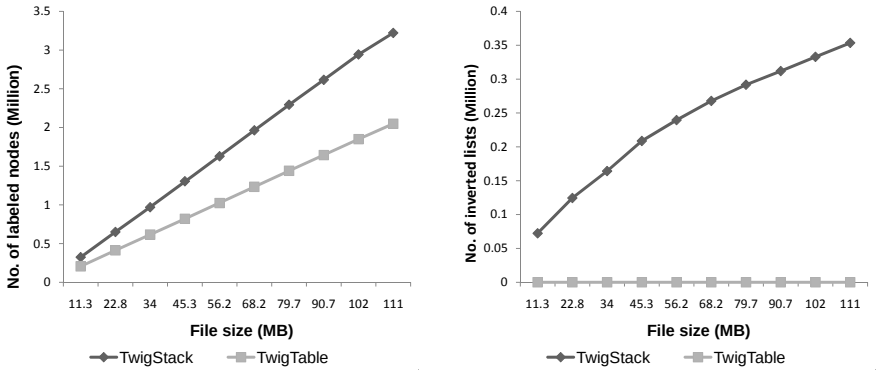
Space management We test the space issues in document parsing, including the number of labeled nodes in memory, and the number of inverted lists and the number of tables maintained on disk. We parse the NASA, the DBLP, and a 110MB XMark data using the two approaches. The result is shown in Fig. 16.

Data	Number of Labeled Nodes			Number of Inverted Lists		Number of Tables	
	TwigStack	TwigTable	Saving	TwigStack	TwigTable	TwigStack	TwigTable
NASA	997,987	532,963	46.6%	121,833	68	N.A.	39
DBLP	6,771,148	3,736,406	44.8%	388,630	37	N.A.	26
XMark	3,221,925	2,048,193	36.4%	353,476	79	N.A.	43

Fig. 16. Number of labeled nodes, inverted lists and tables in *TwigStack* and *TwigTable*

This result validates our analysis in Section 4.4 about the reduction of labeled nodes in memory and the reduction of inverted lists. In *TwigTable*, the relational tables are built based on different types of properties, so the number of tables is

limited to the number of different property types. We also use 10 sets of XMark data, whose sizes vary between 11MB and 110MB, to further demonstrate the superiority of *TwigTable* in space management. The experimental result is shown in Fig. 17. We can see that the number of labeled nodes is scaled to the document size for both approaches, and *TwigTable* always manages less labeled nodes. The number of inverted lists is scaled to the size of document in *TwigStack*, whereas this number is a constant in *TwigTable*. For a large data set it is not practical to handle the tremendous number of inverted lists using *TwigStack*.



(a) Number of labeled nodes (b) Number of inverted lists

Fig. 17. Space management comparisons

Data	Total size for indexes (MB)					
	TwigStack			TwigTable		
	Inverted Lists	Tables	Total	Inverted Lists	Tables	Total
NASA	462	0	462	9	31	40
DBLP	1290	0	1290	30	119	149
XMark	1040	0	1040	17	47	64

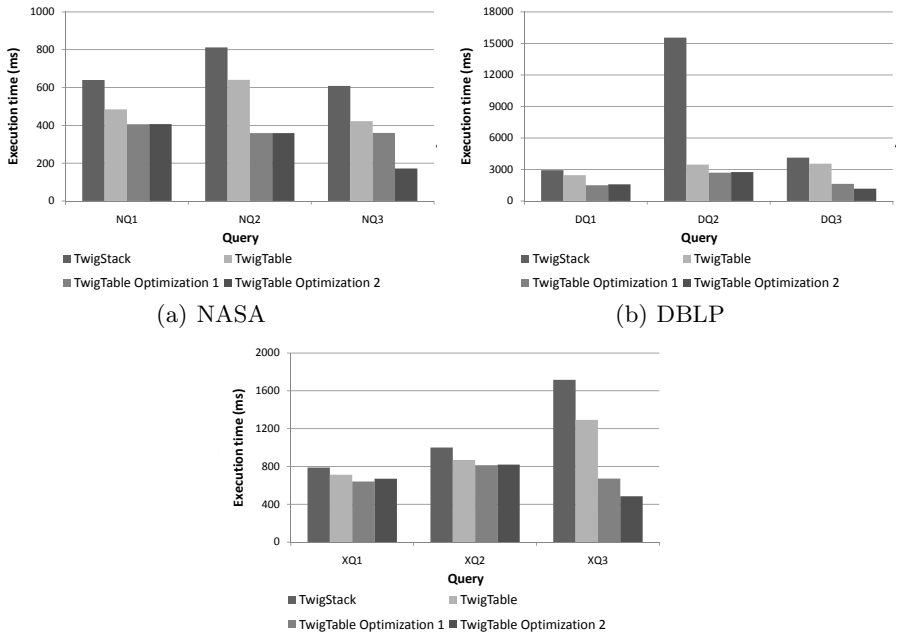
Fig. 18. Index sizes in *TwigStack* and *TwigTable*

We further analyze the total space usage to store indexes of the two approaches, as shown in Fig. 18. We simply use a separate file to represent each inverted list on disk, for both approaches. The inverted lists are further indexed by a B^+ tree automatically by the file system, so that during query processing the relevant inverted lists can be quickly addressed. From the result, we can see that *TwigStack* needs a large size of space to manage inverted lists. We further investigate it and find that actually the total size of inverted lists is not very large, which is similar to the total size used by *TwigTable*. However, because of the tremendous number of inverted lists, the extra structures, e.g., the B^+ tree, built by the file system to index these inverted lists are quite large in size.

In contrast, although *TwigTable* maintains table indexes in addition to inverted lists, the total size is still much smaller.

Query performance We used the NASA, the DBLP and a 110MB XMark data set for query performance comparison. We compare *TwigStack* with our original *TwigTable* algorithm, as well as Optimization 1 and Optimization 2. As mentioned earlier, we can infer the object information in an XML document. Although the inference may not be semantically correct, it will not affect the correctness of the result. In the two optimizations, we use such inference to build object/property tables and object tables. We do not test Optimization 3.

The inverted lists and relational tables are stored on disk and loaded into memory as needed during query processing. As mentioned above, inverted lists are stored as separate files and indexed by a B^+ tree to ensure the high performance in inverted list access. *TwigTable* actually adopts *TwigStack* (with the same implementation) to perform structural search, after content search with table selection. The execution time of *TwigTable* and its optimizations include the I/O and CPU costs to access relational tables to perform content search and the cost on structural search. The comparison result is shown in Fig. 19.



(a) NASA (b) DBLP

(c) XMark

Fig. 19. Execution time by *TwigStack* and *TwigTable* without optimizations, with Optimization 1 and with Optimization 2 in the three XML documents

From the result we can see that for all queries, *TwigTable* outperforms *TwigStack*. The reason is that *TwigTable* performs content search first to simplify the query pattern before structural joins, thus gaining better overall per-

formance. The result also proves that the overhead on table selection will not affect the benefit gained from twig pattern simplification. *TwigStack* performs very badly on DQ2. In the DBLP data, there are a lot of numeric values. To process DQ2, *TwigStack* has to combine the labels in all the inverted lists with a number name greater than 1979, based on document order. To load and merge these inverted lists is costly. However, in *TwigTable* this step is replaced by table selection, thus it is more efficient. We can see, though for small document (e.g., NASA), *TwigStack* is not very slow to perform range search, when the amount of labels in numeric inverted lists is large, *TwigStack* will be inefficient to load and merge the labels.

For all the queries, Optimization 1 works better than *TwigTable*. The reason is that Optimization 1 reduces one more level up to the original twig pattern query. Similarly, the query processing performance is further improved.

Comparing Optimization 1 with Optimization 2, we can see that for single-predicated queries there is no obvious difference. For some queries, Optimization 2 is even slightly worse than Optimization 1. The reason is that the combined object table is larger than object/property table, and then there may be more I/Os to load tuples for object table. However, for multi-predicated queries, e.g., the queries Q3, Q6 and Q9, Optimization 2 has better performance, because Optimization 2 performs content search for all the value comparisons on the same object at the same time. This again proves our analysis in Section 4.4.

7 Conclusion

In this paper, we propose a semantic approach *TwigTable* to solve different kinds of content problems raised in existing approaches for twig pattern query processing. Unlike *TwigStack* and its subsequent algorithms, our approach uses semantic tables to store values in XML document and avoids the management of a tremendous number of inverted lists for different values. During query processing, we perform content search first to reduce the size of relevant inverted lists, and rewrite the query to reduce the number of structural nodes and the number of structural joins in it. Then, we match the simplified pattern with size reduced inverted lists to the document. We also show that the attempt of using hybrid indexes (inverted list and relational table) can easily and efficiently process queries across multiple twig patterns.

Our approach is a semantic approach because the relational tables are initially built based on the semantics of property. With more semantics on objects and relationships, we propose three optimization techniques to further improve the tables and enhance efficiency of query processing. In particular, (1) if each property's associated object is known, we can change property table to object/property table in Optimization 1; (2) if we know certain properties belong to the same object, we can combine the object/property tables to be object table in Optimization 2; (3) if the relationship between objects is known, we can introduce relationship tables to precisely store the property values of relationships in Optimization 3. To summarize, as more semantic information is known, we

can further optimize tables and get better performance. Furthermore, when an output node of a query corresponds to a property type, after finding an occurrence of a twig pattern in the document, our algorithm can easily extract actual value to answer the query from relational tables, whereas existing approaches need more work to convert labels into values by accessing documents again.

Our work brings a new perspective to incorporate relational approach into native approach to manage and query XML data. Processing queries involving ID references [31] or queries across multiple documents becomes possible in our approach, as our approach takes value-based join to link different twig patterns involved in a complex query. We will further investigate how to generate optimal query plans for queries involving both structural joins and table joins.

References

1. http://www.cs.washington.edu/research/xmldatasets/data/nasa/nasa.xml
2. http://www.sybase.com/products/databasemanagement/sqlanywhere
3. S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In Proc. of ICDE, 2002, pp. 141-154.
4. A. Berglund, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML Path Language (XPath) 2.0. W3C Working Draft (2003)
5. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query. W3C Working Draft (2003)
6. P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: a cost-based approach to XML storage. In Proc. of ICDE, 2002, pp. 64-75.
7. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In Proc. of SIGMOD, 2002, pp. 310-321.
8. T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In Proc. of SIGMOD, 2005, pp. 455-466.
9. Y. Chen, S. B. Davidson, C. S. Hara, and Y. Zheng. RRXS: redundancy reducing XML storage in relations. In Proc. of VLDB, 2003, pp. 189-200.
10. A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen. Community information management. In IEEE Data Eng. Bull., vol. 29, no. 1, 2006, pp. 64-72.
11. D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. In IEEE Data Eng. Bull., vol. 22, no. 3, 1999, pp. 27-34.
12. G. Gou, and R. Chirkova. Efficienty querying large XML data repositories: a survey. In IEEE Transactions on Knowledge and Data Engineering, vol. 19, no. 10, 2007, pp. 1381-1403.
13. T. Grust. Accelerating XPath location steps. In Proc. of SIGMOD, 2002, pp. 109-120.
14. H. Jiang, H. Lu, and W. Wang. Efficient processing of XML twig queries with OR-predicates. In Proc. of SIGMOD, 2004, pp. 59-70.
15. H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic twig joins on indexed XML documents. In Proc. of VLDB, 2003, pp. 273-284.
16. C. Li, and T. W. Ling. QED: a novel quaternary encoding to completely avoid re-labeling in XML updates. In Proc. of CIKM, 2005, pp. 501-508.

17. T. W. Ling, M. L. Lee, and G. Dobbie. Semistructured database design (web information systems engineering and Internet technologies series). Springer-Verlag, 2004.
18. Z. Liu, and Y. Chen. Identifying meaningful return information for XML keyword search. In Proc. of SIGMOD, 2007, pp. 329-340.
19. J. Lu, T. Chen, and T. W. Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In Proc. of CIKM, 2004, pp. 533-542.
20. J. Lu, T. W. Ling, C. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In Proc. of VLDB, 2005, pp. 193-204.
21. S. Navathe, S. Ceri, G. Wiederhold and J. Dou: Vertical partitioning algorithms for database design. In ACM Transactions on Database Systems, vol. 9, no. 4, 1984, pp. 680-710.
22. S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov. Indexing XML data stored in a relational database. In Proc. of VLDB, 2004, pp. 1146-1157.
23. P. R. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prufer Sequences. In Proc. of ICDE, 2004, pp. 288.
24. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: limitations and opportunities. In Proc. of VLDB, 1999, pp. 302-314.
25. A. Spink. A user-centered approach to evaluating human interaction with web search engines: an exploratory study. In Information Processing & Management, vol. 38, no. 3, 2002, pp. 401-426.
26. I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In Proc. of SIGMOD, 2002, pp. 204-215.
27. F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies. In SIGMOD Record, vol. 31, no. 1, 2002, pp.5-10.
28. TreeBank. Retrieved from University of Washington Database Group. 2002.
29. H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A Dynamic index method for querying XML data by tree structures. In Proc. of SIGMOD, 2003, pp. 110-121.
30. H. Wu, T. W. Ling, and B. Chen. VERT: a semantic approach for content search and content extraction in XML query processing. In Proc. of ER, 2007, pp. 534-549.
31. H. Wu, T. W. Ling, G. Dobbie, Z. Bao, and L. Xu. Reducing graph matching to tree matching for XML queries with ID references. In Proc. of DEXA, 2010.
32. XMark. An xml benchmark project. http://www.xml-benchmark.org.
33. L. Xu, T. W. Ling, H. Wu, and Z. Bao. DDE: From Dewey to a Fully Dynamic XML Labeling Scheme. In Proc. of SIGMOD, 2009, pp. 719-730.
34. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. In ACM Trans. Internet Techn., vol. 1, no. 1, 2001, pp. 110-141.
35. C. Yu, and H. V. Jagadish. Efficient discovery of XML data redundancies. In proc. of VLDB, 2006, pp. 103-114.
36. T. Yu, T. W. Ling, and J. Lu. Twigstacklistnot: A holistic twig join algorithm for twig query with NOT-predicates on XML data. In Proc. of DASFAA, 2006, pp. 249-263.
37. C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In Proc. of SIGMOD, 2001, pp. 425-436.