VERT: a semantic approach for content search and content extraction in XML query processing

Huayu Wu, Tok Wang Ling, and Bo Chen

School of Computing, National University of Singapore
3 Science Drive 2, Singapore 117543
{wuhuayu, lingtw, chenbo}@comp.nus.edu.sg

Abstract. Processing a twig pattern query in XML document includes structural search and content search. Most existing algorithms only focus on structural search. They treat content nodes the same as element nodes during query processing with structural joins. Due to the high variety of contents, to mix content search and structural search suffers from management problem of contents and low performance. Another disadvantage is to find the actual values asked by a query, they have to rely on the original document. In this paper, we propose a novel algorithm *Value Extraction with Relational Table* ($VERT$) to overcome these limitations. The main technique of $VERT$ is introducing relational tables to store document contents instead of treating them as nodes and labeling them. Tables in our algorithm are created based on semantic information of documents. As more semantics is captured, we can further optimize tables and queries to significantly enhance efficiency. Last, we show by experiments that besides solving different content problems, $VERT$ also has superiority in performance of twig pattern query processing compared with existing algorithms.

1 Introduction

XML plays an important role in information exchange nowadays. As a result, how to process queries over XML documents efficiently attracts lots of research interests. In most XML query languages (see, e.g. [2][3]), the queries are expressed as *twig* patterns. Finding all appearances of a twig pattern in an XML document is a most significant operation in XML query processing.

Normally an XML query includes structural search and content search. E.g. in a query

$$book[author = \text{“}Jack\text{”}]/title$$

$book[author]/title$ is a structural search and $author = \text{“}Jack\text{”}$ is a content search. Most of existing works only focus on processing structural search efficiently and very few of them pay high attentions on contents. Content problems include how to properly manage contents, how to efficiently process content search during query processing and how to extract contents to answer the queries.

E.g. $TwigStack$ [4] and its subsequent algorithm $TwigStackList$ [9] are optimal for processing path and twig pattern queries, but they all suffer from content problems when they process queries with content predicates, because they treat contents the same as other element nodes and perform structural search for the whole query. Although some algorithms like [12] and [11] use subsequence matchings to avoid problems on content search , they still suffer from other problems such as cost for content result fetching in XML databases.

In this paper, we propose a new algorithm to solve different kinds of content problems in twig pattern query processing. Our contribution can be summarized as follows:

- We propose a new algorithm, namely *Value Extraction with Relational Table (VERT)*. Our approach combines value contents to related element or attribute nodes and only assigns label to the element or attribute. Instead of organizing tremendous number of streams for different contents, e.g. streams for value content '33', 'Gehrke' and so on for document in Fig. 1(a), we adopt relational tables to store value contents together with their associated elements or attributes. In this point of view, we can reduce the number of labeled nodes and also we can solve the management problem of contents raised in previous algorithms.

- Content search can be accomplished efficiently by SQL queries on corresponding relational tables with proper indexes. Furthermore, after finding the appearances of a twig pattern we can easily get the desired value contents from tables. As a result, our approach need not consider the document storage for final results.

- Relational tables are created based on semantic information captured in documents. As more semantics is captured, we can further optimize the tables and queries to achieve better performance, as shown in Section 5.

- Besides solving content problems, we also present experimental results to show the superiority of $VERT$ and subsequent optimizations over previous algorithms in performance of twig pattern query processing.

The rest of the paper is organized as follows. We first describe some related works in Section 2. After that we discuss background knowledge and motivation for our work in Section 3. The $VERT$ algorithm with two semantics based optimizations is presented in Section 4. We present the experimental results in Section 5 and conclude our work in Section 6.

2 Related Work

In early work, Zhang et al. [14] proposed a $multi - predicatemergejoin$ algorithm based on (DocId, Start, End, Level) containment labeling of a XML document. Later an improved stack-based structural join algorithm is proposed

by Al-Khalifa et al. [1]. These two algorithms, as well as most of prior works decomposed a twig pattern into a set of binary relationships, e.g. parent-child and ancestor-descendant relationships. Then twig pattern matching could be done by matching each binary relationship and combining these basic binary matches. The main problem of such approaches is that 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. [4] proposed a holistic twig join algorithm, *TwigStack*, which could avoid producing a large size of intermediate result. However, this algorithm is only optimal for a twig pattern with only ancestor-descendent relationships. There are many subsequent works [9] [8] [5] [10] [7] [13] to optimize *TwigStack* in terms of I/O, or extend *TwigStack* for different kinds of problems. In particular, Lu et al. [9] introduced a *list* structure to make it optimal for queries containing parent-child relationships. However, all these existing works only focus on structural search. For content search they either treat content node the same as element node, or consider how contents are stored and perform a separate operation on content search. *ViST* [12] and *PRIX* [11] transform twig pattern queries into sequences and perform subsequence matchings for query processing. They can solve problems on content search, but they still do not pay attention to fetching content results of twig pattern queries, and are not efficient in structural search comparing with *TwigStack*.

3 Background and Motivations

3.1 Data model and twig pattern

XML documents are commonly modelled as ordered trees, in which tree nodes represent tags, attributes or text values, and edges represent element-subelement, element-attribute, element-content or attribute-value pairs. We call these binary relationships parent-child relationships (denoted by "/"). Queries normally appear to be twig patterns. A twig pattern is a small tree whose nodes stand for tags, attributes or contents in a document. Different from the parent-child relationship in XML tree, edges in twig query can also be ancestor-descendent relationships (denoted by "//") which stand for that some other nodes may appear on the path between the two nodes connected by "//".

Given a twig pattern query Q, finding all the occurrences in an XML tree T is the main operation for query processing. A match of Q in T is identified by a mapping of nodes and edges from Q to T such that query node predicates are satisfied by corresponding nodes in the document and the relationships between query nodes are satisfied by corresponding relationships between nodes in the document.

3.2 Motivations

TwigStack and its supplementary works are optimal for twig query processing in many cases. In this section, we take *TwigStack* as an example and discuss

some drawbacks of existing algorithms regarding to contents, which motivate our research.

Similar as most existing algorithms, *TwigStack* processes content search in the same way as structural search. The problems regarding to contents in *TwigStack* can be summarized as follows:

1. In *TwigStack*, all the nodes including elements, attributes and contents in an XML tree are labeled and the labels are organized in streams. When we build streams for contents, stream management is a problem. Consider a bookstore document shown in Fig. 1(a). There are a large number of books and each of them has a unique ISBN number. For each ISBN number there is a stream, e.g. a stream for '0-07-123057-2', another stream for '0-07-124650-9' and so forth. The problem is how to manage the tremendous number of streams. When a query in Fig. 1(b) is issued, it is time consuming to get all the streams with numeric names which are greater than 15. Although we can organize streams using B^+ tree, after finding all the corresponding streams, to combine labels in them by document order is also time consuming.

2. Streams for contents do not have semantic meanings. This may cause additional checking. When the query is interested in books with price of 20, structural search scans the stream T_{20}. Since in T_{20} we do not differentiate *price* and *quantity*, we need check all the labels in this stream though many of them stand for *quantity* and definitely do not contribute to final answer.

3. When we issue a query over an XML document, what we need is not all the twig pattern appearances represented as tuples of labels, but the content results of that query. Like in the query example above, after finding a certain number of twig pattern appearances which contain element $ISBN$ and *title*, we need to find their actual values. Since value contents are stored as stream names and it is not practical to get them using labels, they have to move into the document again. That is relevant to how to store and manage XML document and is not negligible.

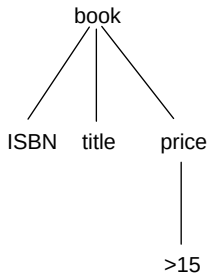
Our motivation is to avoid all these content problems raised in existing algorithms. After that, twig pattern queries can be processed more efficiently not only in content search, but also in entire execution.

4 VERT Algorithm

Some elements or attributes in XML documents describe certain properties of their parent elements, e.g. *title*, *price* are properties of *book* in the bookstore document. We use term *property* for such element or attribute, and term *object* for the element described by *property*. In this section, we first present $VERT$ algorithm to handle content problems and improve efficiency on content search. Tables in $VERT$ store relationships between properties and their values. Then in Section 4.4 we present another two approaches to optimize $VERT$ using

```
<book_store>
  <book>
    <ISBN>0-07-123057-2</ISBN>
    <title>Database Management Systems</title>
    <author>Ramakrishnan</author>
    <author>Gehrke</author>
    <price>33</price>
    <quantity>20</quantity>
  </book>
  <book>
    <ISBN>0-07-124650-9</ISBN>
    <title>Introduction to Database Systems</title>
    <author>Bressan</author>
    <author>Catania</author>
    <price>17</price>
    <quantity>12</quantity>
  </book>
  ......
</book_store>
```

(a) An XML document



(b) A twig pattern query

Fig. 1. An example of XML document and twig pattern query

semantic information. Tables in these optimizations store relationships between objects and their properties with values.

4.1 XML document parsing in VERT

In our first approach, we use tables to store labels of properties and their values. When we parse an XML document, we only label elements and attributes, and put these labels into corresponding streams in document order. Contents in document are not labeled, instead we put them in relational tables together with labels of their *property* nodes. We adopt interval encoding labeling scheme in our approach (see [6]). The detailed algorithm $Parser$ is presented as follows.

There are three major steps when we parse an XML document: labeling the elements, constructing streams for different types of elements or attributes, and inserting each pair of property and value content into relational tables. SAX reads the documents to transform each tag and content into event and line 3 captures the next event if there are more events in SAX stream. Based on different types of events, different operations are performed accordingly. Line 4-16 are executed if the event e is a starting element. In this case, the first 2 steps are processed. The system first constructs an object for this element and assigns a label to it. It then puts the label into the stream for that tag. A stack S is used to temporarily store the object so that when an ending tag is reached, the system can easily know on which object the operation will be executed. At line 9-14, the system analyzes the attributes for this element if any. Based on the same operating steps, it labels the attributes and puts labels into streams. The attribute values are treated in the same way as element contents. Line 17-18

Algorithm 1 Parser

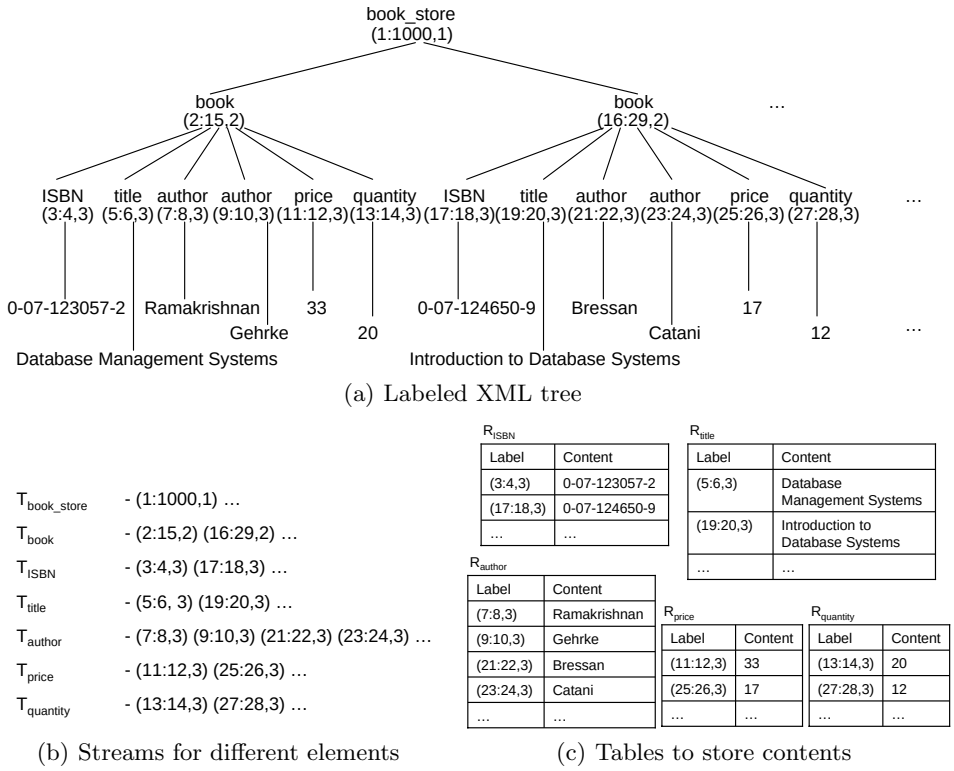
```
1: initialize Stack S
2: while there are more events in SAX stream do
3:    let e = next event
4:    if e is a start tag then
5:       //step 1: label elements
6:       create object o for e
7:       assign label to o
8:       push o onto S
9:       for all attributes attr of e do
10:         //we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

is the case that the event is a content type. Then the content is simply bound to the top object in S for further insertion used. When the event is an ending element in line 19-25, last step is processed, which is popping the top object from the S and inserting the label together with contents into the table for that object. A set of example tables are shown in Fig. 2(c). They are property-value tables. The name of the tables are the property names and each table contains two fields, the label of the property node and value content.

Example 1. Consider the XML data shown in Fig. 1(a), *Parser* assigns labels to tags and the resulting labeled tree is shown in Fig. 2(a). Then all the labels belonging to the same type of element in XML tree will be passed to the same stream by document order as shown in 2(b). The contents in document together with their parent elements will be stored in corresponding relational tables according to the type of parent elements. Fig. 2(c) shows the resulting tables in this example.

4.2 Query processing with *VERT*

Twig pattern query processing involving contents is composed of two parts. First we analyze and rewrite the query. During this part, for each leaf node which is



(a) Labeled XML tree

(b) Streams for different elements

(c) Tables to store contents

Fig. 2. An example of labeled XML tree with resulting streams and tables by Parser

a value content, a new stream for its parent property node is constructed using the table of that property. In the second part, we process the new query using existing efficient algorithms, e.g. $TwigStack$ in new searching space.

In the main algorithm $VERT$, we first check for the validity of a given query in line 2-4. Validity of a query Q is defined as whether Q is meaningful for processing. Intuitively this validation can be accomplished by checking whether all the content comparisons in query predicates have parent element. If there is some content comparison in Q appearing in ancestor-descendance relationship ('//') with an element, we consider Q is not meaningful and our algorithm rejects such Q. Example 2 shows an invalid query. Line 6-12 recursively handle all the content comparisons in two phases: creating new streams and rewriting the predicates. In detail, Line 7-10 execute SQL selection on corresponding tables based on the content comparison, and then put all the selected labels, which are satisfied with the content comparison, into the new streams. Line 11 rewrites the query in such a way that the content and their respective parent elements or attributes are replaced by a new element which has an identical name as the corresponding new stream. At the end of the algorithm, we use $TwigStack$ or other efficient algorithm to process the new query with new streams.

Algorithm 2 VERT

```
1: //check the validity of queries
2: if The query Q is not valid then
3:    reject Q
4: end if
5: //step 1: construct new streams and new queries
6: while there are more content comparisons in predicates of Q do
7:    let c be the next content comparison, and p be its parent element or attribute
8:    create a new stream X_{p'} for p
9:    select the labels based on content c from the table T_p for p
10:   put the selected labels into X_{p'}
11:   rewrite the predicates such that replace sub-structure p/c by p'
12: end while
13: //step 2: process new queries in new streams
14: process the rewritten Q using existing efficient algorithms like TwigStack
```

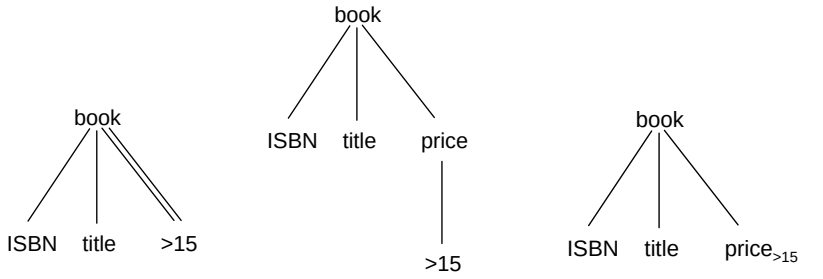
Example 2. Consider the twig pattern query in Fig. 3(a). The value node with content comparison '>15' only has an ancestor instead of a parent. In this case we are not sure whether the query wants to get the books with price greater than 15, or with the quantity greater than 15. As a result, this twig pattern query is considered invalid and by the line 2-4, $VERT$ rejects this query.

Example 3. The twig pattern query in Fig. 3(b) is valid. By $VERT$ we first find the predicate with content comparison, $price > 15$ in this case. In line 7-10, we execute SQL in table R_{price} to get all the labels of element *price* having value content greater than '15'. Since all the records in database are inserted in document order, we can directly add resulting labels into a new stream $T_{price'}$, which contains all the labels for *price* with value greater than 15. Then we rewrite the twig pattern query where the substructure with node *price* and its child node '>15' is replaced by $price'$. To clearly explain $price'$ in the new query, we use $price_{>15}$ in Fig. 3(c). Finally we process $TwigStack$ on the new query using $T_{price'}$ for node $price_{>15}$.

4.3 Analysis of VERT

In this section, we will analyze our algorithm in four points of view: the management of data including labeled nodes and streams, content extraction for both predicates and final results, the size of streams to be searched and the number of structural joins during query processing.

Data management $VERT$ combines contents and their parent elements together, and avoids labeling content nodes separately. Suppose an XML document is a full tree with height of h and each element has k children on average. Then the number of labeled elements is $(k^{h-1}\text{-}1)/(\text{k-}1)$ and the number of contents is k^{h-2}. When the document is large, the proportion of contents to the sum of elements and contents is around (k-1)/(2k-1). In



(a) Invalid query (b) Valid query (c) Rewritten query of (b)

Fig. 3. Invalid and valid twig pattern queries

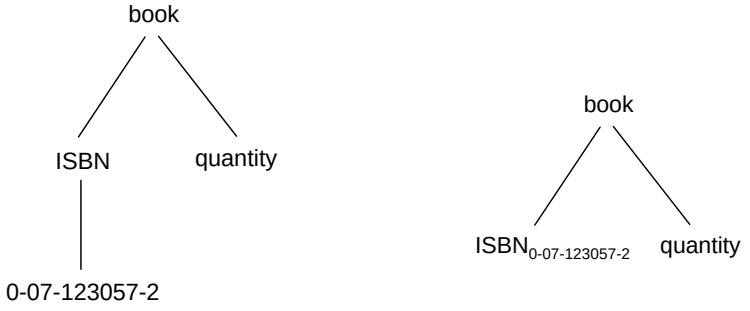
our algorithm, since contents are not labeled, the number of labeled nodes in memory will be reduced nearly by half for large documents and then the size of each stream will also be significantly narrowed down. Furthermore, since large variety of contents are ignored, the number of streams for different types of labeled nodes is limited to the number of element types. So the management of tremendous number of streams in previous work as mentioned in Section 3 can be solved.

Value extraction Consider the XML document in Fig. 1(a) and a query in Fig. 3(b). When we extract the content '15' to answer this query, in previous approaches we need to move into the stream for content '15'. However, the stream for '15' contains different semantic labels like the *price* of a book and the *quantity* of a book. To mix them together will cause unnecessary search. Instead of searching in streams, $VERT$ handles contents in semantic tables. In this case, we just move into the table for *price* and avoid searching for content '15' under *quantity*. Furthermore, after getting all the appearances of $ISBN$ and *title* tags which satisfy the constraint, we aim to find the value contents under these tags. Previous approaches have to move into document again to fetch them because the streams for such contents cannot contribute to final result extraction. This depends how XML documents are stored and is not negligible. $VERT$ can be very efficient to get the desired content results without considering document storage because all the contents are stored in tables instead of streams and we can directly get these contents through SQL operations on tables. As a result, relational tables are not only helpful in content search, but also usable to get desired contents based on the labels found.

Stream searching reduction Pre-processing contents is essential to reduce the size of streams. Consider the query that we want to find the quantity for a book with '$ISBN = 0 - 07 - 123057 - 2$' on the bookstore document. If the number of different books is b, the size of stream for element $ISBN$ is also b in previous approaches, as shown in Fig. 4(a). Then we need

$O(b)$ to scan all the labels in $ISBN$ stream. $VERT$ processes selection in advance, such that the new stream for $ISBN$ is created based on content '0-07-123057-2'. That means the new stream has only 1 label inside since $ISBN$ is the key for books. Fig. 4(b) shows the rewritten query and size of new stream. $T_{ISBN'}$ is the new stream for element $ISBN$, and in Fig. 4(b) we use $ISBN_{0-07-123057-2}$ to explain $ISBN'$. So when the selectivity of an element is high, like in this example, $VERT$ also has high superiority to previous algorithms because it significantly reduces the searching in stream.

Structural joins reduction There are two factors driving the high performance of $VERT$. One is searching space reduction as mentioned above and the other factor is number of structural joins reduction. Still consider the example in Fig. 4. The rewritten query has only two parent-child relationships need structural joins, while the original query has three. As we know structural join is an expensive operation, the reduction of structural joins leads a higher performance. Optimizations to further reduce size of streams and number of structural joins will be discussed in next section.



(a) Original query with corresponding stream size $|T_{book}|$=b and $|T_{ISBN}|$=b

(b) Rewritten query for (a) with corresponding stream size $|T_{book}|$=b, and stream size for node $ISBN_{0-07-123057-2}$=$|T_{ISBN'}|$=1

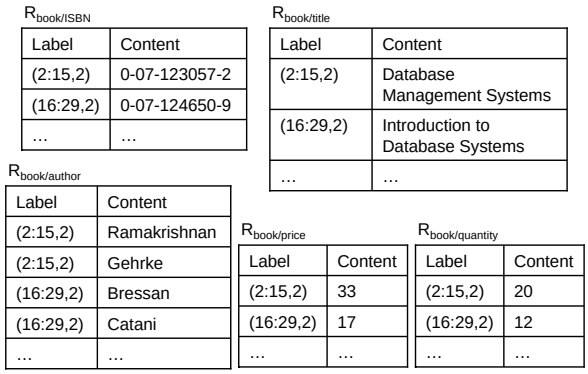
Fig. 4. Original and rewritten query examples in VERT

4.4 Optimizations for VERT

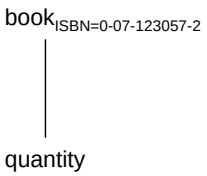
Tables in $VERT$ store label of each property and its value content. This approach differentiates contents by properties. However, there are still more semantics can be captured. With more semantic information, we can improve the performance by further reducing size of streams and number of structural joins in most documents. This motivates two approaches of optimizations for $VERT$.

Observation 1: Generally, after knowing the value on a certain property of an object, most queries want to find that object and then get some other properties of that object. For the query in Fig. 4, with the $ISBN$ value we need to find corresponding *book* and get the *quantity* of that book. After $VERT$ rewriting the query, the size of stream for $ISBN$ is significantly reduced. However the size of stream for *book* is still b, which means we need to search all the b labels in *book* stream although we know there is only one matches with the label in $ISBN$ stream. As a result, we can further rewrite the query to get the object *book* directly from property $ISBN$ value.

Optimization 1: Instead of storing labels of property nodes and their value contents, we can put the labels of objects with property values into tables. E.g. in the bookstore document we put value contents for $ISBN$, *title* and so forth with labels of corresponding *book* in *object/property* tables as shown in Fig. 5(a). The 'Label' field of each table stores the label of object *book* and the following 'Content' corresponds the value contents of different properties in different tables, e.g. in $R_{book/ISBN}$ 'Content' in each tuple is the $ISBN$ value of the book with 'Label' in the same tuple. The query in Fig. 4(a) is rewritten accordingly, as shown in Fig. 5(b), where $T_{book'}$ is the new stream for element *book* and $book_{ISBN=0-07-123057-2}$ is to explain $book'$. In new tables, we can directly select the label for book based on ISBN number in $R_{book/ISBN}$ without considering tags for element $ISBN$. Now we not only reduce the size of T_{book}, but also reduce the number of structural joins to be 1. So we can get a higher performance when we execute the new query.



(a) Tables in VERT optimization 1



(b) Rewritten query for Fig 4(a) with stream size of $book_{ISBN=0-07-123057-2}$= $|T_{book'}|$=1

Fig. 5. Tables and queries in VERT optimization 1

However, this optimization may lose order information in some cases. If we want to get all the authors' names of a certain book, since we ignore the element

author and get all the name contents from $R_{book/author}$, we cannot differentiate the order in document. This limitation can be solved by adding ordinal number to different contents if the order is important.

Observation 2: There are some queries with multiple predicates on a certain element. E.g. a query on the bookstore document: find the ISBN number of the book with title 'Database Management Systems' and price of 33. To answer this query, $VERT$ with optimization 1 needs to find the books with title 'Database Management Systems' and books with price of 33 separately and join them to get results. With semantic information, we know *title* and *price* are properties of object *book*. If we have a table for this object which contains both of the properties, books satisfying these two constraints can be found directly and intermediate results can be avoided.

Optimization 2: A simple idea is to pre-merge tables in optimization 1 based on the same objects. But for multi-value properties, like *author* in our example, it is not practical to merge it with other properties. The information on multi-value properties can be found in document schema. After knowing this, we can merge all the single-value properties of an object into one table and keep tables for multi-value properties remain as what they are in optimization 1. The resulting tables for bookstore document is shown in Fig. 6. In R_{book}, each label of book is stored with all the single-value property contents of that book. When we process queries with multiple predicates on a certain object, we can do selection in that object table using these predicate constraints in one time. In this way, we can even simplify the query and prune intermediate results.

R_{book}

Label	ISBN	Title	Price	Quantity
(2:15,2)	0-07-123057-2	Database Management Systems	33	20
(16:29,2)	0-07-124650-9	Introduction to Database Systems	17	12
...	...	...	...	...

$R_{book/author}$

Label	Author
(2:15,2)	Ramakrishnan
(2:15,2)	Gehrke
(16:29,2)	Bressan
(16:29,2)	Catani
...	...

Fig. 6. Tables in VERT optimization 2

Declarations: Optimizations of $VERT$ are based on semantics captured from schema or document, or even declared by document owners. Generally, the more semantic information known, the further our algorithm can be optimized and the better performance can be achieved.

5 Experiments

In this section, we present experimental results on the performance of twig pattern search under $VERT$ algorithms with and without optimizations, which are introduced in section 4, and $TwigStack$. Final result extraction for each

query can be done simply by selection in corresponding tables in our approach, however, in other algorithms it depends on the database implementation. The comparison for final result extraction is not included in our experiments.

5.1 Experimental Settings

Implementation and Hardware: We implemented all algorithms in Java. The experiments were performed on a 3.0GHz Pentium 4 processor with 1G RAM under OS of Windows XP.

XML Data Sets: We used three real-world and synthetic data sets for our experiments: NASA, DBLP and XMark. NASA is a 25 MB document with complex DTD schema. DBLP data set is a 127MB fragment of DBLP database. The characteristic of this data set is simple DTD schema and large data sources. We also used XMark benchmark data with size of 110MB.

Queries: We selected three meaningful queries for each data set. All the queries chosen contain predicates with content comparison, since content predicates appear in most practical queries. Generally, there are three types of queries: queries with predicates on equality comparison, queries with predicates on range comparison and queries with multiple predicates on different comparisons. The queries are shown in Table 1.

Query	Data Set	XPath Expression
Q1	NASA	//dataset//source/other[/date/year>'1919' and year<'2000']/author /lastName
Q2	NASA	//dataset/tableHead[//field/name='rah']//tableLinks //title
Q3	NASA	//dataset/history/ingest[/date[/year>'1949' and year<'2000'] [/month='Nov'][/day>'14' and day<'21']]/creator /lastName
Q4	DBLP	/dblp/article[/author='Jim Gray']/title
Q5	DBLP	//proceedings[/year>'1999']/isbn
Q6	DBLP	//inproceedings[/title='A Flexible Modelling Approach for Software Reliability Growth'][/year='1987'][/author='Sergio Bittanti'] /booktitle
Q7	XMark	//regions/africa/item[/mailbox/mail/from='Libero Rive'] /description
Q8	XMark	//item[//mail/date>'Sep']/location
Q9	XMark	//item[/location='United State'][/mailbox/mail[/date= '02/11/1999'][/to='Aamer Krolokowski']]/description

Table 1. Experimental queries

5.2 Experimental Results and Analysis

Our experiments mainly compare the stream management and total execution time between $TwigStack$ and our approaches. The implementation of $TwigStack$ adopts B^+ tree to organize streams, which ensures high performance of content stream management. The number of labeled nodes and number of streams to be managed for the three data sets under the two approaches are shown in Table 2. This result validates the analysis about the data management in last section.

Data Set	Number of Labeled Nodes		Number of Streams	
	TwigStack	VERT	TwigStack	VERT
NASA	997,987	532,963	121,833	68
DBLP	6,771,148	3,736,406	388,630	37
XMark	5,215,282	2,048,193	353,476	75

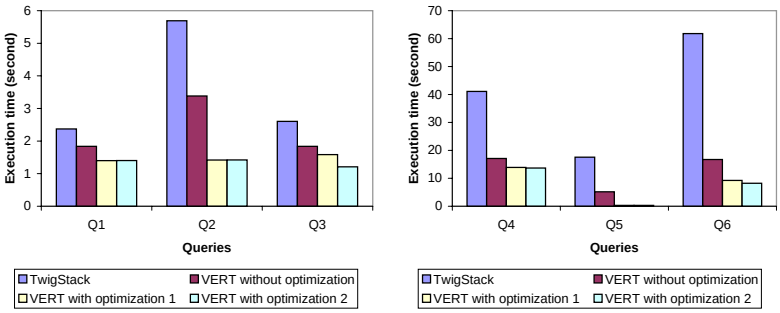
Table 2. Number of labeled nodes and streams using $TwigStack$ and $VERT$

The experimental results of execution time for the three data sets are shown in Fig. 7. From the results, we can see the execution time reduction is significant for all the queries in DBLP document. This is in accord with our analysis in section 4.3, that is our approach works quite well for such XML document that has simple schema but large data sources. In DBLP document, there are only several types of data like proceedings, thesis, articles and so forth. There are large quantity of works under each type. The properties of each work type that appear as sub-elements in document are mostly the same and depth of the data tree is 3. As a result, for DBLP data, when we rewrite the query to reduce the query depth, we prune tremendous number of unnecessary tag checkings. Q2 in NASA data set is another example for the reason why $VERT$ has higher performance than other approaches. The tag 'name' appears quite frequently in document with different semantic meanings, however, in Q2 what we are interested is only the field name. Instead of scanning all the 'name', our approach can move into field names directly using semantic tables. In this way, the execution time can be significantly reduced.

Comparing with optimization 1 and optimization 2 of $VERT$, we can see from the experimental result that for single-predicate queries there is no obvious difference. However, for multi-predicate queries, optimization 2 has a better performance as shown in Q3, Q6 and Q9. This again proves our analysis in Section 4.4.

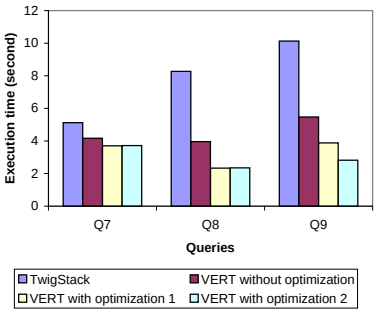
6 Conclusion and future work

In this paper, we propose a novel algorithm $VERT$ to solve different content problems raised in existing algorithms. Unlike $TwigStack$ and its subsequent algorithms, our approach uses semantic tables to do content search, and then



(a) NASA

(b) DBLP



(c) XMark

Fig. 7. Execution time by TwigStack and VERT without optimizations, with optimization 1 and with optimization 2 in three XML documents.

avoids the management of tremendous number data streams. Besides, $VERT$ can efficiently extract contents for predicate comparisons during query processing. Experimental results show that our method is much more efficient than $TwigStack$ for queries with content comparison as predicates. To answer the query, our method need not consider how the document stored in database. Instead, we can directly get the content results from tables.

One direction for future research to improve our algorithm is to discover more semantics in XML document and combine the semantic information into relational tables. Queries can be processed more efficient based on semantic tables and query rewriting by reducing unnecessary searches, number of structural joins and intermediate results. Also, our relational approach gives a new scheme to relate XML query processing algorithms to XML databases.

References

1. 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)

2. A. Berglund, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML Path Language (XPath) 2.0. W3C Working Draft (2003)
3. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query. W3C Working Draft (2003)
4. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In Proc. of ACM SIGMOD (2002)
5. T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In Proc. of SIGMOD Conference (2005)
6. T. Grust. Accelerating XPath location steps. In Proc. of SIGMOD Conference (2002)
7. H. Jiang, H. Lu, and W. Wang. Efficient processing of XML twig queries with OR-predicates. In Proc. of SIGMOD Conference (2004)
8. H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic twig joins on indexed XML documents. In Proc. of VLDB Conference (2003)
9. 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)
10. 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 Conference (2005)
11. P. R. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prufer Sequences. In Proc. of ICDE (2004)
12. 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 Conference (2003)
13. 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)
14. C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In Proc. of ACM SIGMOD (2001)