Accelerating XML Query Processing on Views

Yin-Fu Huang and Yu-Hsien Cho
National Yunlin University of Science and Technology
123 University Road, Section 3, Touliu, Yunlin, Taiwan 640, R.O.C.
E-mail: huangyf@yuntech.edu.tw, http://mdb.csie.yuntech.edu.tw

Keywords: XPath, labeling schemes, materialized views, T-Bitmap, tag index, value index, navigation, twig query

Received: April 13, 2016

With the widespread use of the eXtensible Markup Language (XML), more and more applications store and query XML documents in XML database systems. Thus, how to efficiently process a query and find the specified patterns conforming the query from XML documents is a crucial issue. In this paper, some processing methods are employed on XML documents to improve document retrieval. First, a materialized view is built from an original document for each query. Then, on each materialized view, auxiliary structures such as T-Bitmap and indexes are also built to further accelerate query processing. Finally, four experiments are conducted to show the superiority of the proposed approach.

Povzetek: Predstavljena je metoda za hitrejše iskanje po bazah XML dokumentov.

1 Introduction

Since XML (eXtensible Markup Language) was widely used to exchange data over the web, more and more applications store and query XML documents in XML database systems. Different from other data formats, an XML document is composed of elements and values with a nested structure, and could be modeled as a tree structure. XPath and XQuery are the standard XML query languages proposed by W3C. They can be used to describe patterns with specified predicates on multiple elements with tree structured relationships. However, how to efficiently process a query and find the specified patterns conforming the query from XML documents is a crucial issue.

In the past, different methods have been proposed in querying XML documents. One of research directions was to build materialized views on XML documents. The goal is to reduce the number of visited nodes during tree traversing by searching from the root of a materialized view, rather than from the root of an original XML document tree. Another research direction was to construct index or access methods to query XML documents for facilitating query processing. In this paper, we integrate the methods from these two research directions as our motivation for accelerating XML query processing on views. The reason is that the performance of a materialized view is better than a non-materialized view because not only these data can be accessed without re-materialization, but also they can be fetched faster by building indexes on these data beforehand. Besides, a materialized view is usually used in accessing a large amount of data, such as data warehouse applications, in support of management's decision-making process through OLAP queries, almost read operations In short, the motivation is for decision makers to accelerate XML query processing in a data warehouse.

In summary, we highlight the contributions of this paper as follows:
1) In this study, we build **materialized views** from an XML document for each query to reduce the search space of queries, and also build **auxiliary structures** such as T-Bitmap and indexes to further accelerate query processing.
2) **Comprehensive experiments** are conducted to verify the superiority of the proposed approach.
3) **The space vs. time issue** is explored when multiple materialized views are integrated together to save the space.

The remainder of this paper is organized as follows. Section 2 presents the previous work proposed in querying XML documents. In Section 3, basic concepts such as query processing and materialized views on XML documents are introduced. In Section 4, we propose a system architecture consisting of view processing and query processing. In Section 5, four experiments are conducted to show the superiority of our approach. Finally, we make conclusions in Section 6.

2 Previous work

As mentioned in Section 1, one research direction on querying XML documents was to build materialized views on XML documents to reduce the number of visited nodes during tree traversing, thereby leading to faster query processing. Godfrey et al. [1], and Murthy and Banerjee [2] proposed SQL/XML syntax for query processing on views, whereas Halevy [3] and Jayavel et al. [4] proposed various query syntax such as join to handle views and focused on the problem of evaluating XML queries over XML views of relational data. However, users must be familiar with these various query syntax. Katsifodimos et al. [5] considered choosing the

best views to materialize within a given space budget to improve the performance of a query. Roantree and Liu [6] approach is to segment a materialized view into fragments to minimize the effect of view changes. Bonifati et al. [7] presented an algebraic approach for propagating source updates to materialized views. Wu et al. [8, 9] proposed a bitmapped materialized views approach for optimizing XML queries. Gosain et al. [10] provided a survey of materialized view evolution methods, which aims at studying the materialized view evolution in relational databases and data warehouses as well as in a distributed setting. Gosain and Sachdeva [11] drew several conclusions about the status quo of materialized view selection and a future outlook is predicted on bridging the large gaps that were found in the existing methods.

Another research direction was to construct index or access methods to query XML documents, also improving query processing. Some studies investigated constructing index methods to query XML documents [12-15]. Bruno et al. [12] and Jiang et al. [13] used a structure join method to determine element relationships based on the numbering scheme. This method has good performances for an ancestor-descendant axis, but it might fetch useless nodes for a parent-child axis, because all descendant nodes must be accessed to check if they are real children. Therefore, Huang and Wang developed an efficient query processing algorithm for retrieving XML documents [14]. Hsu et al. also proposed a path clustering method based on the concept of summary indexes for the processing of both structural and content queries on XML documents [15]. Karthiga and Gunasekaran [16] used tree-based association rules to mine the semantics from XML documents, which provide information on both the structure and the content of XML documents. The mined knowledge is used to provide the quick answers to queries and an approach called path based indexing is used to improve the speed of data retrieval. Alghamdi et al. [17], and Thi Le et al. [18] proposed approaches to optimizing twig queries by utilizing the semantics/constraints defined in XML schemas. Furthermore, Ordonez focused on the optimization of linear recursive queries in SQL [19]. Subramaniam and Haw [20] proposed an XML labeling scheme that helps quick determination of structural relationship among XML nodes and supports dynamic updates without relabeling nodes in case of update occurrences. Belgamwar et al. [21] follows an upside down approach which explicitly stores the values and only reconstructs the internal nodes, if needed. As a solution, they proposed a compressed internal storage format for native XML database systems where the inner structure of the gathered documents is virtualized. Ferro and Silvello [22] introduced a new paradigm where traditional approaches based on traversing trees are replaced by a brand new one based on basic set operations which directly return the desired subtree, avoiding to create it. Tudor [23] proposed an optimization model for XML data processing based on a heuristic algorithm to extract data from XPath views.

3 Basic concepts

3.1 XML documents

XML is a markup language which was proposed by W3C in 1996. The main purpose of the standard language is to provide data descriptions and data exchanges across different platforms. Like other markup languages, the contexts of XML are declared between start and end tags; however, especially different from others, the tags can be flexibly defined by users to describe data, and furthermore XML is supported in different platforms and systems. That is why it becomes the most common format for data exchanges.

An XML document is with a nested structure, and it could be represented as a rooted, ordered, and labeled tree structure. Figure 1 and Figure 2 illustrate an XML document and its corresponding tree representation, respectively. In the document, there is a unique root element called “root” and one of the descendant elements, called “Book”, has seven child element nodes; i.e., Title, Chapter, Para, Author with an attribute node “Id”, Publisher, Name, Email, and their texts. The symbols as shown in Figure 2 are circles, rectangles, and triangles; they represent elements, texts, and attributes, respectively.

```
<?xml version="1.0" standalone="yes"?>
<root>
  <store>
    <Books category="Technology">
      <Book>
        <Title>How to know XML</Title>
        <Chapter>
          Introduction to XML
          <Para>Your First XML</Para>
        </Chapter>
        <Author Id="Q345">John</Author>
        <Publisher>
          <Name>XML tech</Name>
          <Email>John@hpdiy.zzn.com</Email>
        </Publisher>
      </Book>
      <Book>
          ⋮
      </Book>
    </Books>
  </store>
</root>
```

Figure 1: XML document.

3.2 XPath

XPath (XML Path Language) is an expression language for addressing and querying an XML document. In XPath expressions, each step is separated by "/" and contains three components: **axis**, **node test**, and **predicate**. **Axis** defines the relationship to be followed in the document tree. **Node test** defines what kind of nodes

is required. **Predicate** is optional and provides the capability to filter nodes, according to selection criteria.

Given an XPath example "//child::Publisher [child::Name='XML tech'] /child::Email" , it is to get the email of the publisher whose name is "XML tech". When navigating the XML document, it must start from the root element "root", then the descendant node "Publisher". Beneath "Publisher", we search the child nodes to find the node called "Email". Besides, during the search, it must have a child node called "Name" whose text matches with the specified predicate "XML tech". In general, the example above can be abbreviated to "//Publisher [Name ='XML tech'] /Email".

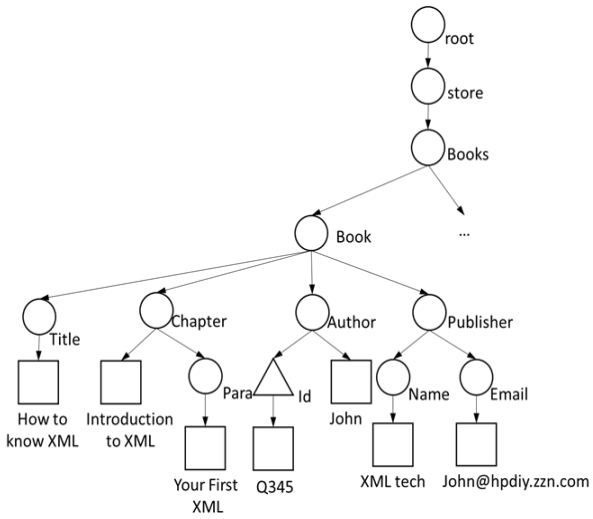

Figure 2: XML document tree.

3.3 Labeling schemes

One of the major query searches is to determine the relationships between nodes. In order to determine element relationships quickly, several different labeling schemes have been proposed. O'Connor and Roantree categorized labeling schemes into containment schemes, prefix schemes and prime number schemes [24]. Here, labeling schemes are classified into prefix-based ones and region-based ones (or containment schemes).

Dewey code [25] is a prefix-based labeling scheme that records the position information of a node, according to the path from the root to the node. For example, Dewey-id of node "Para" is 1.1.1.1.2.2, and indicates that we can get node "Para" if we search alone the path (the first node of level 1, the first node of level 2, the first node of level 3, the first node of level 4, the second node of level 5, the second node of level 6). Besides, since (1.1.1.1.2) is the prefix of (1.1.1.1.2.2), the relationship between node "Chapter" (1.1.1.1.2) and "Para" (1.1.1.1.2.2) can be deduced as a parent-child one. However, the drawback of the prefix-based labeling scheme is its lengthy Dewey codes, especially when the levels of an XML document tree are too deep.

The region-based labeling scheme [12] is another numbering scheme. The label contains three elements (start, end, level) where the start value and end value forms a region. The region of an upper-level node (i.e., ancestor or parent) must cover those of lower-level nodes (i.e. children or descendants). In other words, if node A covers node B, then *A.start* < *B.start* and *B.end* < *A.end*. Besides, the level value represents the node level in a document tree. With the coverage information, we can determine the relationships between nodes quickly. As for the labeling, we can label each node by traversing an XML document tree in a depth-first search way.

3.4 XML document storage

An XML documents can be stored in a few different forms, such as in flat files, in relational databases, and in native XML databases. For an XML document to be stored in flat files, we need to parse the files in advance before accessing them. Although it is the simplest form, the parsing time would be very lengthy when the XML document size is too large. Besides, it also incurs multi-user access and concurrency control problems. For an XML document to be stored in relational database, since the XML document is a tree structure, it must use some middleware to translate the XML format into relational tables. Besides, when querying the XML document, it is also necessary to translate a query into an SQL statement, and execute join operations repeatedly among different relation tables, so that it exposes lower efficiency. Native XML databases aim to provide complete XML document storage and manipulation. Different from other database systems, native XML databases use an XML document as a basic unit of storage, and defines an XML model used to store and retrieve XML documents.

3.5 Materialized views

A view is a virtual and derived table defined by users for facilitating to express a complicated query. Rather than physically stored as parts of a database, a view definition is merely recorded by the database system. It is evaluated only when a user issues a query involving this view. However, a materialized view is the one which is physically stored in the database, in addition to its definition. Absolutely, the performance of a materialized view is better than a non-materialized view because not only these data can be accessed without re-materialization, but also they can be fetched faster by building indexes on these data beforehand. Thus, a materialized view is usually used in accessing a large amount of data, such as a data warehouse or in business intelligence applications, where we need to take more time to query them. A data warehouse is a subject-oriented, integrated, time-variant, and nonvolatile collection of data in support of management's decision-making process. It can be accessed by decision makers through OLAP queries, almost read operations. In short, our design is for decision makers to accelerate XML query processing in a data warehouse.

In this paper, based on native XML databases, we use materialized views to query required data from an original document. Here, a materialized view can be defined using the "CREATE MATERIALIZED VIEW"

function and an XPath expression. For the materialized views on an original document, we build auxiliary files and construct indexes using numbering schemes to avoid unnecessary sub-tree traversal, thereby improving the navigation efficiency of a query.

4 System architecture

4.1 Overview

In order to achieve faster query processing on the views defined in a native XML database, we propose a system architecture consisting of an offline phase and an online phase, as shown in Figure 3. In the offline phase called view processing, we build view-relevant structures such as T-Bitmap and indexes to accelerate later query processing. In the online phase called query processing, the system can promptly respond to view-based queries, utilizing the T-Bitmap and indexes built beforehand.

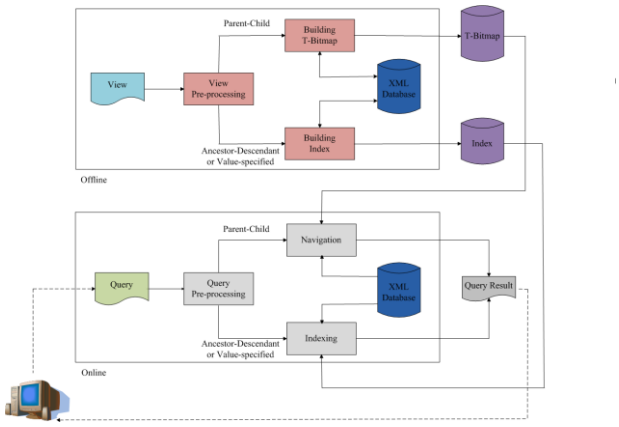

Figure 3: System architecture.

4.2 View processing

In this section, the motivation of view materialization is introduced first. Then, we build relevant structures such as T-Bitmap and indexes on materialized views to further accelerate query processing.

4.2.1 View pre-processing

Usually, an Xpath expression is used to address and query an XML document. However, for the query execution, the system always searches an XML document tree from the root. When a query is frequently executed, the system performance would be degraded since a large amount of unnecessary sub-tree traversal cannot be avoided. For the query with an Xpath expression as shown in Figure 4, we can define a materialized view beforehand, which is rooted from node "Books" with an attribute "category" matching with the specified predicate "Technology", as shown in Figure 5. Then, the materialized view can be created from the original document, as shown in Figure 6. Thus, rather than traversing the original document tree always from the root, the system only needs to search the materialized view, thereby improving the navigation efficiency of the query.

```
XPath:/root/store/Books[@category="Technology"]
/Book[Title="How to know XML"]/Publisher
/Email='John@hpdiy.zzn.com'
```

Figure 4: Query with an XPath expression.

```
CREATE MATERIALIZED VIEW mv AS(
SELECT extract(sys_nc_rowinfo$,
'/root/store/Books[@category="Technology"]')
FROM XMLTABLE);
```

Figure 5: View definition.

```
<?xml version="1.0" standalone="yes"?>
  <Books category="Technology">
    <Book>
      <Title>How to know XML</Title>
      <Chapter>
        Introduction to XML
        <Para>Your First XML</Para>
      </Chapter>
      <Author Id="Q345">John</Author>
      <Publisher>
        <Name>XML tech</Name>
        <Email>John@hpdiy.zzn.com</Email>
      </Publisher>
    </Book>
    <Book>
      <Title>Small World</Title>
      <Chapter>
        Q&A
        <Para>The One</Para>
      </Chapter>
      <Author Id="A854">Jimmy</Author>
      <Publisher>
        <Name>Network</Name>
        <Email>Jimmy@hpdiy.zzn.com</Email>
      </Publisher>
    </Book>
          ⋮
  </Books>
```

Figure 6: Materialized view.

Before building T-Bitmap and indexes on a materialized view to further accelerate query processing, we must determine the relationships between nodes (i.e., parent-child axes and ancestor-descendant axes) in a materialized view using the region-based labeling scheme as mentioned in Section 3.3. We traverse a materialized view and label nodes in a depth-first search way. When a node is visited first, its start value is created; when we leave the node, the end value is labeled. After traversing the whole materialized view, all the nodes in the view are completely labeled as shown in Figure 7.

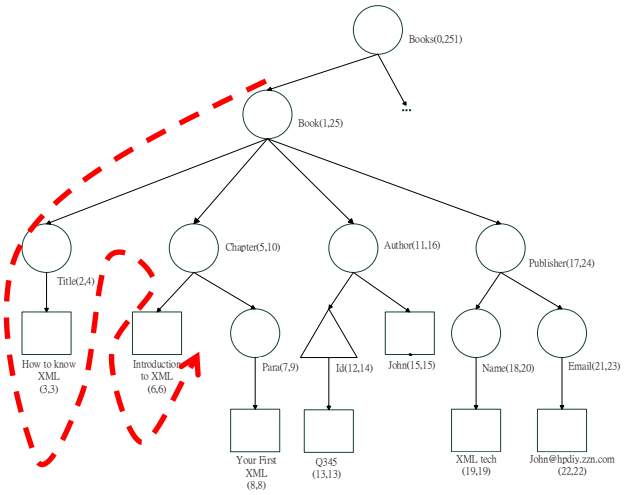

Figure 7: Labeling nodes in a depth-first search way.

4.2.2 Building T-Bitmap

T-Bitmap is a bit string type, which is used to record what descendant nodes are beneath a current node. First, a dictionary recording the positions in T-Bitmap and the corresponding tags is created, as shown in Table 1. Then, the T-Bitmap value on each node can be calculated using OR operators. For an example as shown in Figure 8, the T-Bitmap of node "Publisher" can be calculated by combining the T-Bitmaps of "Name", "Email", and itself using OR operators.

Table 1: Dictionary: positions in T-Bitmap and corresponding tags.

Position	0	1	2	3
Tag	Books	Book	Title	Chapter
Position	4	5	6	7
Tag	Para	Author	Id	Publisher
Position	8	9	-	-
Tag	Name	Email	-	-

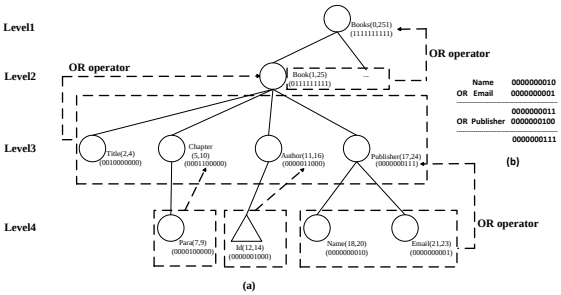

Figure 8: Combing T-Bitmaps using OR operators.

4.2.3 Building index

Here, we use labeling codes to build two kinds of index trees; i.e., tag index trees and value index trees. To illustrate the tag index construction, we extend the storage model in Figure 7. As shown in Figure 9, we can see a lot of nodes with the same tag names but with the different labeling codes; e.g., node "Email(21, 23)" and "Email(46, 48)". We can build the index tree of each tag using the start values in labeling codes as keys and the well-known B+-tree algorithm, as shown in Figure 10 where the pointers of a leaf node in the tag index tree indicate the positions of corresponding nodes in the materialized view. For the query with an XPath: "Book//Email", when processing the current node "Book(26, 50)", we can use the tag index tree of "Email" to locate each leaf node by following the dotted path, and find out node "Email(46, 48)" covered by node "Book(26, 50)".

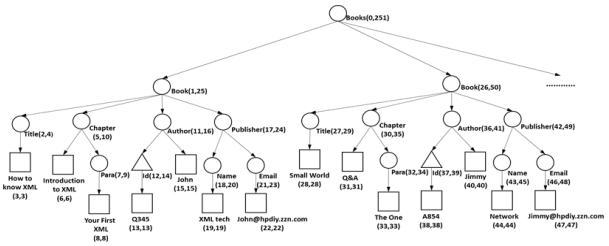

Figure 9: Extended storage model.

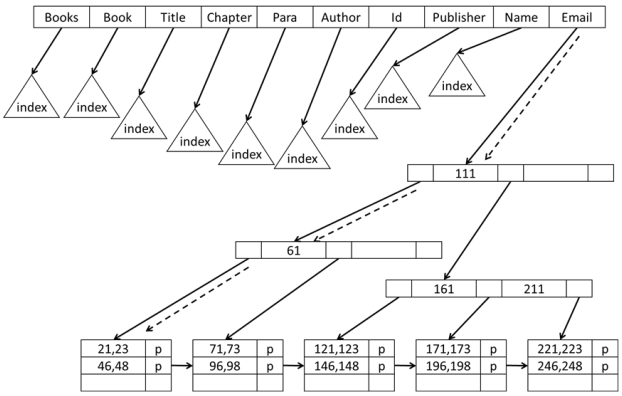

Figure 10: Tag index tree.

Besides, we can also build a value index tree according to the text values of nodes in the document, as shown in Figure 11. The construction method is the same as that used to build tag index trees. However, we generate only one value index tree for each materialized view, and the records of a leaf node are with the [text, start, pointer] format where the pointers also indicate the positions of corresponding nodes in the materialized view.

4.3 Query processing

In this section, the query transformation based on a materialized view is introduced first. Then, according to different axes specified in the transformed query, we make use of the T-Bitmap and indexes built in the view processing to accelerate query processing. Finally, we also introduce subsequent processing for a query specifying the particular predicate in an Xpath expression.

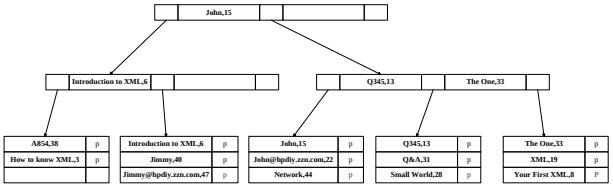

Figure 11: Value index tree.

4.3.1 Query pre-processing

After materialized views are created, a query should be transformed based on its corresponding materialized view. For the query as shown in Figure 4, its Xpath expression (traversing from node root) is transformed into a new one (traversing from node "Books") as shown in Figure 12.

```
SELECT extract(sys_nc_rowinfo$,'/Books
/Book[Title="How to know XML"]/Publisher
/Email=' John@hpdiy.zzn.com '')
FROM mv;
```

Figure 12: Query transformation based on a materialized view.

Before utilizing T-Bitmap and indexes to accelerate query processing, we must deal with parent-child axes and/or ancestor-descendant axes specified in the transformed query. We execute navigation or indexing according to different axes, and recursively check nodes in the document tree.

4.3.2 Navigation

For a parent-child axis, we use a navigation way to search nodes in the document tree. Here, T-Bitmap can be used to avoid unnecessary search during the navigation, since it provides the information whether result nodes are beneath the current processing node. For a query "/R/A/C" as shown in Figure 13, we can use an AND operator to determine whether node C is beneath the current processing node. First, for current node R, Query(11010)^R(11111)=Query(11010) indicates node C is beneath node R. Then, for the next node A1, Query(01010)^A1(01100)≠Query(01010) indicates node C cannot be beneath node A1, and we do not need to search the sub-tree rooted at node A1. Next, for node A2, Query(01010)^A2(01110)=Query(01010) indicates node C is beneath node A2. Finally, we recursively check nodes in the document tree until node C is found.

4.3.3 Indexing

For a query "/A//B· · ·" specifying an ancestor-descendant axis as shown in Figure 14, although we can also use T-Bitmap to search node B, the search based on a parent-child axis would go through a lot of unnecessary intermediate nodes. Therefore, we use indexes to directly search a descendant node, instead of using T-Bitmap. As mentioned in Section 4.2.3, we use the start value in the labeling code of the ancestor as the key to search the tag index tree of the descendant. Then, we check whether the descendant node is covered by the ancestor node; if yes, we fetch the descendant node and proceed to parse the query downward.

4.3.4 Subsequent processing

In this section, we investigate the processing for the query specifying a particular predicate. One is the query specifying values, and another is the twig query.

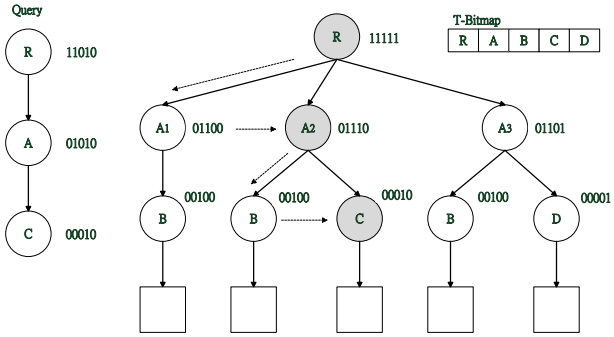

Figure 13: Navigation using T-Bitmap.

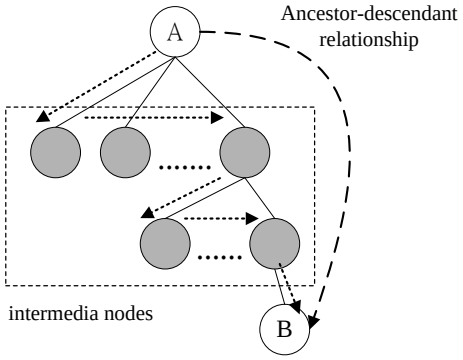

Figure 14: Unnecessary intermediate nodes.

4.3.4.1 Query specifying values

For a query "/Books/Book[Author = 'Jimmy']/Publisher = 'XML tech' ", two values are specified for filtering nodes in the document. As shown in Figure 15, we use the value index tree as mentioned in Section 4.2.3 to find out value nodes "Jimmy(12)", "XML tech1(7)", and "XML tech2(15)", and then put them into their corresponding queues, respectively. When processing node "Book" in the query, we fetch node "Book1", and find that node "Jimmy(12)" is not covered by node "Book1"; i.e., [2,9] < 12. Then, for the next node "Book2", although node "Book2" covers node "Jimmy(12)", node "XML tech1(7)" is not covered by node "Book2". Then, we choose the next value node "XML tech2(15)" and do the same inspection. Finally, we find out node "Book2" is the result node.

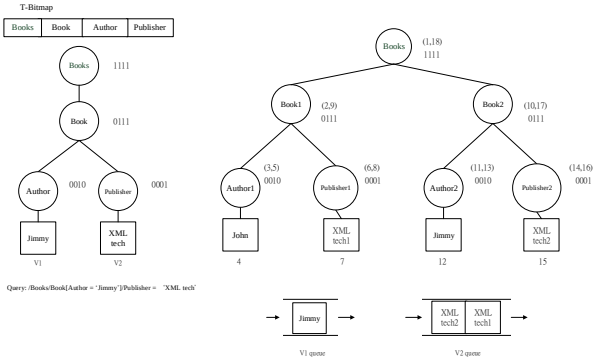

Figure 15: Value node processing.

4.3.4.2 Twig Query

Besides, for a query "/Book[Publisher]/Author" as shown in Figure 16(a), we can also process it in the same way as done in Section 4.3.2. As shown in Figure 16(b), two solutions "Book, Publisher, Author1, Jim" and "Book, Publisher, Author2, Jimmy" exist in the document. They can be found by 1) merging Path1 and Path2, and 2) merging Path1 and Path3, as shown in Figure 16(c).

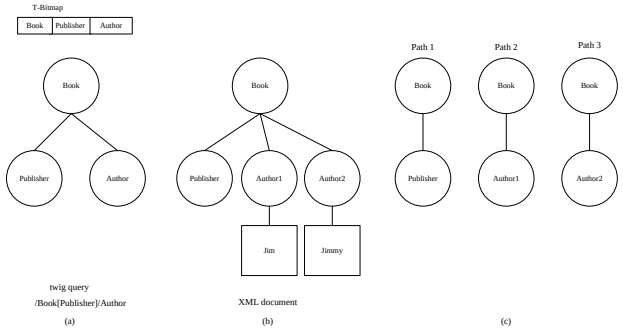

Figure 16: Twig query processing.

5 Experiments

In this section, four experiments are conducted to show the superiority of our approach proposed in this paper. These experiments are written in Java (JDK1.7) and conducted on an Intel Pentium4 3GHz CPU with 3G main memory in Windows 7. In the first experiment, we present the comparisons among different methods. In the second experiment, we investigate the effect of query types on different methods, especially on our method. In the third experiment, we use synthesis documents to analyze our method, and try to find some characteristics. In the last experiment, we address the space vs. time issue if multiple materialized views can be integrated together to save the space; in other words, more than one query would search from a materialized view.

5.1 Comparisons among different methods

In this experiment, we compare the search ways in different methods as shown in Figure 17. The first method is the original search way which is always from the root of a document tree. The second method was proposed by Godfrey et al. [1], which searches from the root of a materialized view. The last method is ours which also searches from the root of a materialized view, but with the aid of auxiliary data structures.

To fairly compare with the method proposed by Godfrey et al., an XML benchmark available on the XMark site is used in the experiment, which has data size 113.794MB and 1,513,518 nodes. Also, we follow the similar query types and comparison ways used in the experiments conducted by Godfrey et al. As shown in Table 2, there are twelve different types of queries tested in the experiment. To contrast with the searching ways as shown in Figure 17, the columns as shown in Table 3 are 1) query types, 2) searching time on the original tree, 3) view creation time, 4) searching time using Godfrey et al.' method, and 5) searching time using our method. The

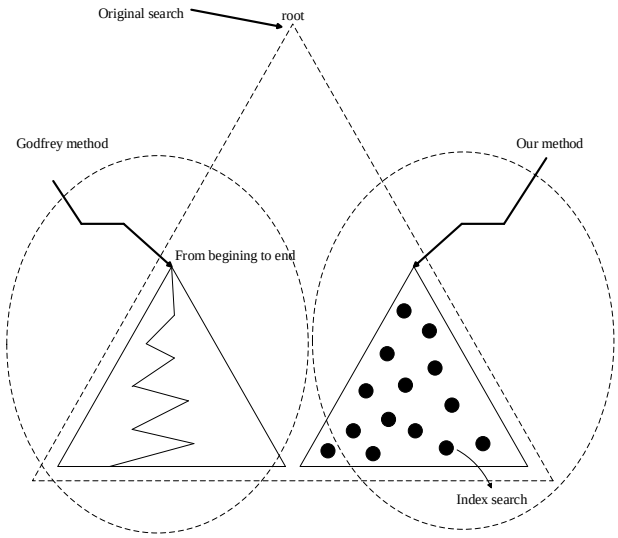

Figure 17: Search ways in different methods.

experimental results show that our method performs much better than the Godfrey et al.' method in all the queries, especially for long-path and twig queries Q3, Q7, Q8, Q9, and Q11. This is why our method uses the T-Bitmap and index structures to accelerate query processing.

5.2 Effect of query types on different methods

In this experiment, we use the same XML benchmark as the first experiment, but different queries as shown in Table 4. For the searching axis, Q1 and Q2 are based on the parent-child axis, Q3 and Q4 on the ancestor-descendant axis, and the others on mixed axes. For the query types, Q1, Q3, Q5, Q6, and Q7 are path queries, whereas the others are twig queries. Furthermore, Q1, Q3, Q4, Q5, Q6, Q7, and Q10 are with value predicates.

In this experiment, as shown in Table 5, our method is still the best one among different methods in all the queries. Even taking the worst case Q8 as an example, our method is 206 times faster than the original way, and 88 times faster than the Godfrey et al.' method. The reason is, as shown in Table 6, the number of nodes visited for Q8 in our method is only 1/28 times of the original way, and is only 1/10 times of the Godfrey et al.' method.

For the path queries (i.e., Q1, Q3, Q5, Q6, and Q7), both execution time of the Godfrey et al.' method and ours is less than one second. For the twig queries (i.e., Q2, Q8, and Q9), they cost more execution time than the path queries since a large amount of nodes are visited. However, for the similar twig queries (i.e., Q4 and Q10), they do not cost much execution time since only a very small amount of nodes are required to visit. In summary, the advantages of our method are 1) when dealing with twig queries, we only need to check T-Bitmap and skip an entire sub-tree if not matched, and 2) when dealing with the queries with value predicates, we can use the value index tree to achieve efficient processing.

Table 2: Twelve kinds of queries.

Q1	/site/regions/europe/item/mailbox/mail
Q2	/site//item/mailbox/mail
Q3	/site//africa/item/description/parlist/listitem
Q4	/site//person/profile/interest[@category]
Q5	/site//person/profile[age]/interest[@category="category620"]
Q6	/site//person/profile[contains(age,"18")]/education
Q7	/site//open_auction[@id="open_auction5"]//date
Q8	/site//category/description[text]/parlist/listitem
Q9	/site//category/description[text/keyword]/parlist/listitem
Q10	/site/*/*/item/mailbox/mail
Q11	/site//*//africa/item/name
Q12	/site//item/mailbox[count(mail)]

Table 3: Comparisons among different methods.

Time(ms)				
Query	Original tree	[%] View creation	[*] View non-indexed	[#] View indexed
Q1	193718	30131	64980	5752
Q2	195146	32147	84574	20599
Q3	310315	31137	100130	593
Q4	185361	32890	99890	29436
Q5	193474	31147	147344	6865
Q6	191034	29702	65248	9106
Q7	203447	27112	132522	279
Q8	212511	28317	60353	232
Q9	241417	30169	64384	916
Q10	185357	31603	80727	19811
Q11	199274	28417	65059	320
Q12	209115	31731	99283	20687

[%] View creation: Views created for both methods
[*] View non-indexed: Godfrey et al.' method
[#] View indexed: Ours

5.3 Experiments on synthesis documents

In this experiment, six synthesis documents with different fanout are used to analyze our method for three different types of queries as shown in Table 7. Q1 is based on the parent-child axis, Q2 is based on the ancestor-descendant axis, and Q3 is a twig query with three predicates and based on mixed axes. As shown in Table 8, we find that the execution time of each query increases as the fanout increases. Moreover, regardless of the complexity in Q3, it still costs almost the same time as Q1 and Q2 using our method; i.e., its execution time would not increase significantly even if it is a twig query with three predicates. However, for Q3 using the original way and/or the Godfrey et al.' method, their execution time increases seriously as shown in Table 9. Especially for the Godfrey et al.' method, the execution time for fanout30 is 202 times slower than that for fanout5.

5.4 Experiments on space vs. time

In this experiment, we explore the space vs. time issue when multiple materialized views are integrated together. In order to achieve the premise that more than one query can search from a materialized view, we reuse seven queries (i.e., Q3, Q4, Q5, Q6, Q7, Q8 and Q10) as shown in Table 4. Then, we find that 1) Q3, Q6, and Q7 can search from the materialized view built based on Q3, 2) Q4 and Q10 can search from the materialized view built based on Q4, and 3) Q5 and Q8 can search from the materialized view built based on Q8. Thus, after the integration, we have three materialized views for these seven queries. The data space and execution time between no integration and integration are shown in Table 10. Taking the group (Q3, Q6, Q7) as an example, the overall data space is 3+1+3=7(KB) and the total execution time is 3+2+12=17(ms) if each query has its own materialized view; however, after the integration, the data space is only 3(KB), but the total execution time increases to 3+5+26=34(ms).

In order to explore the relationship between data space and execution time for these two strategies (i.e., no-integration and integration), we define two terms: 1) amount ratio for data space and 2) speed ratio for execution time as follows.

Table 4: Different kinds of queries.

Q1	/site/open_auctions/open_auction[@id="open_auction5"]/initial
Q2	/site/open_auctions/open_auction[annotation/author]/bidder/date
Q3	//site//open_auctions//open_auction[@id="open_auction0"]//current
Q4	//person[@id="person0"][creditcard]//watch
Q5	/site/regions//item[@id="item0"]//mail
Q6	//open_auction[@id="open_auction0"]/bidder/date
Q7	/site/open_auctions/open_auction[@id="open_auction0"]/../end
Q8	/site/regions//item[//text/bold]//location
Q9	//closed_auctions/closed_auction[//description/text]/seller
Q10	//people/person[@id="person0"][//business]/name

Table 5: Execution time.

Time(ms)				
Query	Original tree	View creation	View non-indexed	View indexed
Q1	31818	14920	193	8
Q2	68769	15731	37949	1883
Q3	29718	11787	140	3
Q4	30989	10056	103	3
Q5	27976	10705	119	3
Q6	30123	11723	48	2
Q7	32680	12135	139	12
Q8	4344137	222996	1853537	21111
Q9	1560306	246697	201444	16185
Q10	28425	10374	70	2

Table 6: Number of nodes visited.

Nodes			
Query	Original tree	View non-indexed	View indexed
Q1	48005	57	5
Q2	259604	87331	23794
Q3	24001	64	3
Q4	31502	24	8
Q5	34124	50	3
Q6	25960	8	5
Q7	24005	64	3
Q8	1204343	432651	43502
Q9	736217	102736	39003
Q10	25647	24	5

Table 7: Three kinds of queries.

Q1	/root/L1/R
Q2	//root//R
Q3	/root//L1[//R][//Q][//S]

$$amount-ratio_{strategy}=\frac{space(strategy)}{space(original)} \quad (1)$$

$$speed-ratio_{strategy}=\frac{time(strategy)}{time(original)} \quad (2)$$

where *space(strategy)* is the overall data space used in the no-integration or integration strategies, *time(strategy)* is the total execution time required in the no-integration or integration strategies, *space(original)* is the data space of the original document, and *time(original)* is the execution time on the original document.

Then, we can use these two terms to judge which strategy is better for the system performance as follows.

$$\frac{amount-ratio_{integration}}{amount-ratio_{no-integration}}<\frac{speed-ratio_{no-integration}}{speed-ratio_{integration}} \quad (3)$$

or $$\frac{space(integration)}{space(no-integration)}<\frac{time(no-integration)}{time(integration)} \quad (4)$$

For Equation (4), the former term represents that the data space benefits for the integration strategy can be

Table 8: Comparisons for different fanout in our method.

Time(ms)			
	Q1	Q2	Q3
Fanout5	358	326	392
Fanout10	571	554	569
Fanout15	1065	969	1023
Fanout20	1514	1432	1486
Fanout25	2164	2023	2137
Fanout30	2701	2579	2658

Table 9: Comparisons for different fanout in different methods.

Q3 Time(ms)			
	Original tree	View non-indexed	View indexed
Fanout5	143618	419	392
Fanout10	278615	796	569
Fanout15	436672	2700	1023
Fanout20	655059	10815	1486
Fanout25	678379	38339	2137
Fanout30	879231	84779	2658

Table 10: Comparisons between no integration and integration.

No integration	Space(KB)	Time(ms)
Q3	3	3
Q4	1	3
Q5	2	3
Q6	1	2
Q7	3	12
Q8	56226	21111
Q10	1	2
Integration	Space(KB)	Time(ms)
(Q3, Q6, Q7)	3	3+5+26
(Q4, Q10)	1	3+2
(Q5, Q8)	56226	16491+21111

Table 11: Comparisons based on Equation (4).

	(Q3, Q6, Q7)	(Q4, Q10)	(Q5, Q8)
Former term	3/7=0.43	1/2=0.5	56226/56228=1
Latter term	17/34=0.5	5/5=1	21114/37602=0.56

gained (i.e., data space reduced) whereas the latter term represents that the execution time benefits for the no-integration strategy can be gained (i.e., execution time reduced). If Equation (4) with the equal weighting between space and time is true, the integration strategy should be adopted. According to the data taken from Table 10, we can calculate the former term and latter term in Equation (4), as shown in Table 11. From the statistical data, we find that the integration strategy is better for group (Q3, Q6, Q7) and group (Q4, Q10), but the no-integration strategy is better for group (Q5, Q8). Absolutely, different strategies can be adopted for different query groups at the same time to make the system performance in the best status.

6 Conclusions

In this paper, we employ some processing methods on XML documents to improve document retrieval. The goal is to reduce the number of visited nodes during tree traversing, thereby leading to faster query processing. To achieve this goal, we focus on the usage of database views. First, we build a materialized view from an original document for each query. Then, on each materialized view, we also build auxiliary structures such as T-Bitmap and indexes to further accelerate query processing. According to different axes specified in an Xpath expression, we have different techniques to handle them. Finally, through the experiments, we 1) compare the performances among different methods, 2) investigate the effect of query types on them, 3) use

synthesis documents to analyze our method, and 4) address the space vs. time issue if materialized views are integrated together.

References

[1] Godfrey P, Gryz J, Hoppe A, et al. Query rewrites with views for XML in DB2. In: Ioannidis Y, Lee D, Ng R, eds. Proceedings of the IEEE 25th International Conference on Data Engineering, Shanghai, China, 2009. 1339-1350

[2] Murthy R, Banerjee S. XML schemas in Oracle XML DB. In: VLDB Endowment, eds. Proceedings of the 29th International Conference on Very Large Data Bases, Berlin, Germany, 2003. 1009-1018

[3] Halevy Y. Answering queries using views: a survey. Very Large Data Bases Journal, 2001, 10: 270-294

[4] Jayavel S, Jerry K, Eugene S, et al. Querying XML views of relational data. In: VLDB Endowment, eds. Proceedings of the 27th International Conference on Very Large Data Bases, Rome, Italy, 2001. 261-270

[5] Katsifodimos A, Manolescu I, Vassalos V. Materialized view selection for XQuery workloads. In: Fuxman A, eds. Proceedings of ACM SIGMOD International Conference on Management of Data, Scottsdale, Arizona, USA, 2012. 565-576

[6] Roantree M, Liu J. A heuristic approach to selecting views for materialization. Software: Practice and Experience, 2014, 44: 1157-1179

[7] Bonifati A, Goodfellow M, Manolescu I, et al. Algebraic incremental maintenance of XML views. ACM Transactions on Database Systems, 2013, 38: 14:1-14:45

[8] Wu X, Theodoratos D, Wang W H, et al. Optimizing XML queries: bitmapped materialized views vs. indexes. Information Systems, 2013, 38: 863-884

[9] Wu X, Theodoratos D, Kementsietsidis A. Configuring bitmap materialized views for optimizing XML queries. World Wide Web, 2015, 18: 607-632

[10] Gosain A, Sabharwal S, Gupta R. Architecture based materialized view evolution: a review. Procedia Computer Science, 2015, 48:256-262

[11] Gosain A, Sachdeva K. A systematic review on materialized view selection. In: Satapathy S et al., eds. Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications. Advances in Intelligent Systems and Computing, Vol. 515. Springer, Singapore, 2017. 663-671

[12] Bruno N, Koudas N, Srivastava D. Holistic twig joins: optimal XML pattern matching. In: Franklin M J, eds. Proceedings of ACM SIGMOD International Conference on Management of Data, Madison, WI, USA, 2002. 310-321

[13] Jiang H, Wang W, Lu H, et al. Holistic twig joins on indexed XML documents. In: VLDB Endowment, eds. Proceedings of the 29th International Conference on Very Large Data Bases, Berlin, Germany, 2003. 273-284

[14] Huang Y F, Wang S H. An efficient XML processing based on combining T-Bitmap and index techniques. In: Biaz S, Bellaachia A, eds. Proceedings of the IEEE International Symposium on Computers and Communication, Marrakech, Morocco, 2008. 858-863

[15] Hsu W C, Liao I E, Wu S Y, et al. An efficient XML indexing method based on path clustering. In: Alhajj R S, eds. Proceedings of the 20th IASTED International Conference on Modeling and Simulation, Banff, Alberta, Canada, 2009. 339-344

[16] Karthiga D, Gunasekaran S. Optimization of query processing in XML document using TAR and path based indexing. International Journal of Computer Science and Network Security, 2013, 13: 119-127

[17] Alghamdi N S, Rahayu W, Pardede E. Object-based semantic partitioning for XML twig query optimization. In: Barolli L et al., eds. Proceedings of the IEEE 27th International Conference on Advanced Information Networking and Applications, Barcelona, Spain, 2013. 846-853

[18] Thi Le D X, Maghaydah M, Orgun M A, et al. Optimization of XML queries by using semantics in XML schemas and the document structure. In: Lin X et al., eds. Proceedings of the 14th International Conference on Web Information Systems Engineering, Nanjing, China, 2013. 343-353

[19] Ordonez C. Optimization of linear recursive queries in SQL. IEEE Transactions on Knowledge and Data Engineering, 2010, 22: 264-277

[20] Subramaniam S, Haw S C. ME labeling: a robust hybrid scheme for dynamic update in XML databases. In: Ismail M, Ramli N, eds. Proceedings of the IEEE 2nd International Symposium on Telecommunication Technologies, Langkawi, Malaysia, 2014. 126-131

[21] Belgamwar H C, Dhore S M, Rathod P U, Deshmukh S S, Nandanwar G S. Review on storing and indexing XML documents upside down. International Journal for Engineering Applications and Technology, 2015, Manthan-15

[22] Ferro N, Silvello G. Descendants, ancestors, children and parent: a set-based approach to efficiently address XPath primitives. Information Processing & Management, 2016, 52:399-429

[23] Tudor N L. Query optimization against XML data. Studies in Informatics and Control, 2016, 25:173-180

[24] O'Connor M F, Roantree M. Desirable properties for XML update mechanisms. In: Proceedings of the 2010 EDBT/ICDT Workshops, Lausanne, Switzerland, 2010

[25] Lu J, Ling T W, Chan C Y, et al. From region encoding to extended Dewey: on efficient processing of XML twig pattern matching. In: VLDB Endowment, eds. Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, 2005. 193-204