ELSEVIER

Information and Software Technology 45 (2003) 11–22

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

An efficient inverted index technique for XML documents using RDBMS

Chiyoung Seo[a,*], Sang-Won Lee[b], Hyoung-Joo Kim[a]

[a]*School of Computer Science and Engineering, Seoul National University, Sinlim-Dong, Gwanak-Ku, Seoul 151-742, South Korea*
[b]*School of Information and Communication Engineering, SungKyunKwan University, Suwon, South Korea*

Received 16 January 2002; revised 18 May 2002; accepted 1 June 2002

Abstract

The inverted index is widely used in the existing information retrieval field. In order to support containment queries for structured documents such as XML, it needs to be extended. Previous work suggested an extension in storing the inverted index for XML documents and processing containment queries, and compared two implementation options: using an RDBMS and using an Information Retrieval (IR) engine. However, the previous work has two drawbacks in extending the inverted index. One is that the RDBMS implementation is generally much worse in the performance than the IR engine implementation. The other is that when a containment query is processed in an RDBMS, the number of join operations increases in proportion to the number of containment relationships in the query and a join operation always occurs between large relations. In order to solve these problems, we propose in this paper a novel approach to extend the inverted index for containment query processing, and show its effectiveness through experimental results. In particular, our performance study shows that (1) our RDBMS approach almost always outperforms the previous RDBMS and IR approaches, (2) our RDBMS approach is not far behind our IR approach with respect to performance, and (3) our approach is scalable to the number of containment relationships in queries. Therefore, our results suggest that, without having to make any modifications on the RDBMS engine, a native implementation using an RDBMS can support containment queries as efficiently as an IR implementation.

Keywords: Inverted index; XML; Query processing

1. Introduction

1.1. Motivation

As XML has become an emerging standard for information exchange on the World Wide Web [6], a great number of companies and organizations have adopted XML as their basic document format. For example, it is already being used to describe all kinds of cars in motor companies, or to make a library catalog in libraries, or to display various goods on E-Commerce Web sites. As a result, a large volume of XML documents may be generated and a great deal of research needs to be done for extracting information efficiently from XML documents.

Several query languages have been suggested as a tool for extracting information from XML documents [7,8,14,35]. Although it is not obvious which query language will become a standard, the important fact is that containment queries become a crucial part of queries submitted over XML information retrieval systems. We define containment queries as a class of queries based on containment and proximity relationships among elements, attributes, and their contents as in Ref. [46]. At this point, a question arises: how could we support containment queries efficiently? To solve this problem, we consider the inverted index technique, which is widely used in the existing Information Retrieval (IR) field. We therefore extend the inverted index for XML documents and consider two methods for storing the inverted index and handling containment queries as a recent work [46]: using an RDBMS or using an IR engine.

Zhang et al. [46] extended the inverted index for XML documents and compared the implementation on the RDBMS with the IR alternative. However, it has two drawbacks with respect to extending the inverted index. One is that the RDBMS implementation is much worse in terms of performance than the IR implementation. The other is

* Corresponding author. Tel.: +82-28801830; fax: +82-28820269.
E-mail addresses: cyseo@oopsla.snu.ac.kr (C. Seo), wonlee@ece.skku.ac.kr (S.W. Lee), hjk@oopsla.snu.ac.kr (H.J. Kim).

0950-5849/03/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0950-5849(02)00157-X

that when a containment query is processed in the RDBMS, the number of join operations increases as the number of containment relationships in the query increases and a join operation always occurs between large relations.

In this paper, we extend the inverted index in a different way to overcome these problems. In particular, our performance study shows that (1) our RDBMS implementation almost always outperforms the RDBMS and IR implementations in Ref. [46], and (2) our RDBMS implementation is not far behind our IR implementation with respect to performance.

1.2. Advantages of using an RDBMS

There are several advantages in using an RDBMS for storing the inverted index and processing containment queries. Firstly, if we store XML documents as well as the inverted index in the RDBMS, we can build an integrated IR system processing queries to XML documents. Secondly, we can guarantee that the system is stable as more than 20 years have been spent in carrying out research on query execution, query optimization, concurrency control, and recovery technique. Thirdly, it is very easy to build XML query processors handling containment queries on top of the RDBMS. Lastly, most organizations already have an RDBMS installed, so no additional costs are incurred.

1.3. Overview of this paper

This paper is organized as follows. Section 2 discusses related works, and Section 3 describes containment queries. Section 4 explains how to extend the inverted index for handling containment queries, and Section 5 describes how to process containment queries in the RDBMS. Section 6 explains experiments to show the effectiveness of our approach, and finally, Section 7 presents the conclusion.

2. Related work

Several works have been done on handling containment queries, including Refs. [17,46]. These works use the RDBMS to store the inverted index and process containment queries. While Ref. [46] considers the RDBMS and IR implementations and conducts an experiment on each implementation, Ref. [17] only conducts an experiment on the RDBMS implementation.

In Ref. [46], it is shown that while RDBMSs are generally poorly suited for containment queries, under certain conditions they can outperform the IR engine. In addition, Ref. [46] identifies two significant causes that differentiate the performance of the RDBMS and IR implementations: the join algorithms employed and the hardware cache utilization. Therefore, Ref. [46] expects that by combining better join algorithms with better cache utilization, an RDBMS will be able to natively support containment queries efficiently. However, in this paper, we will show that our RDBMS approach is not far behind our IR approach with respect to performance. In the remainder of this paper, we will refer to the approach suggested by Zhang et al. [46] as 2-INDEX approach, since Ref. [46] maintains two different inverted indexes for XML documents.

On the other hand, Florescu et al. [17] extends the inverted index in a different way to 2-INDEX approach and stores it in an RDBMS. However, [17] they do not compare the implementation on the RDBMS with the IR implementation with respect to performance.

Till recently, a number of indexing techniques for semi-structured data and XML data have been suggested. A simplified version of an IR-style text index is used in Lore [29], which integrates text search with semi-structured database, to search strings containing specific text words or groups of text words [20,30]. Li and Moon [28] propose a new system for indexing and storing XML data based on a numbering scheme for elements. Unlike our work these previous indexing schemes do not consider supporting containment queries. There are other indexing techniques for semi-structured data and XML data [4,11,16,26,33]. However, these indexing techniques also do not focus on supporting containment queries.

Considerable works have been done on integrating text searching with relational, object-relational, or object-oriented database systems [5,13,44]. Commercial products include the DB2 Text Extender [25], SQL Server Full-Text Search Service [43], and Oracle InterMedia Text [34]. However, none of these previous works conducts the performance comparison between the RDBMS approach and the IR approach.

Since the emersion of SGML [19], there has been a lot of research on integrating content and structure in text retrieval [1,2,5,31,32,36]. Refs. [9,10,12] deal with containment queries. The significant difference between our approach on containment queries and the previous work is that we mainly concentrate on the implementation of containment algorithms in an RDBMS rather than the development of containment algorithms.

There are a lot of other works using an RDBMS for querying and storing XML documents [15,18,27,37,38,39,40,42,45]. However, these works mainly deal with methods for converting XML documents into relational tables, and vice versa.

Besides, there are several commercial products for storing and querying XML documents; for example, XYZFind [24], Excelon [22], and Tamino [23]. These products, however, which are database systems for XML documents, do not use an RDBMS.

3. Containment queries

Containment queries are a class of queries based on containment and proximity relationships among elements,

```
<books>
  <book>
    <title> Data on the Web </title>
    <author>
      <family>Abiteboul</family><given>Serge</given>
      <family>Buneman</family><given>Peter</given>
      <family>Suciu</family><given>Dan</given>
    </author>
    <summary>
      This book mainly mentions
      <keyword>semisturctured data</keyword> and
      <keyword>XML</keyword>
    </summary>
  </book>
</books>
```

Fig. 1. An example of XML data.

attributes, and their contents [46]. As these containment queries are crucial parts of queries processed by XML IR systems, we should consider finding an effective method for processing containment queries when developing XML IR systems. We will discuss this in further detail in Section 4.

In this paper, we use path expressions, which are similar to those of XQuery [7], in representing containment queries. XQuery is a query language proposed by W3C and designed to be a small, easily implementable language with which queries can be concisely expressed and easily understood. XQuery adopts several features which existing XML query languages and database query languages have. From XPath [8] and XQL [35], XQuery took a path expression syntax suitable for hierarchical documents. From XML-QL [14], it took the notion of binding variables and then using the bound variables to create new structures. From SQL, it took the idea of a series of clauses based on keywords that provide a pattern for restructuring data (the SELECT-FROM-WHERE pattern in SQL). Therefore, it is a new XML query language providing various functions.

Figs. 1 and 2 show an example of XML data and a tree representation of XML data, respectively. We classify basic containment queries to XML documents into four types as in Ref. [46] and use Fig. 1 to show examples of each type.

- *Indirect Containment Query*. A query consisting of indirect containment relationships (predecessor–descendant relationships) among elements, attributes, and their contents.

Example 1. /books//author//'Abiteboul'. When an XML document is represented by means of a tree such as in Fig. 2, the leading "/" indicates that "books" must be a root element and "//" represents a predecessor–descendant relationship. Therefore, this query means retrieving XML documents in which "books" root elements have "author" descendant elements and in turn "author" elements have "Abiteboul" descendant words.

- *Direct Containment Query*. A query consisting of direct containment relationships (parent–child relationships) among elements, attributes, and their contents.

Example 2. /books/book/summary/keyword/'XML'. "books" root elements must have "book" child elements and in turn "book" elements must have "summary" child elements. "summary" elements must have "keyword" child elements, whose content contains "XML". Therefore, this query involves extracting all XML documents satisfying the above conditions.

- *Tight Containment Query*. A query consisting of tight containment relationships among elements, attributes, and their contents.

Example 3. //given = 'Peter'. This query means retrieving XML documents in which "given" elements contain only "Peter" in their contents.

- *k-Proximity Containment Query*. A query based on the proximity between two words in contents.

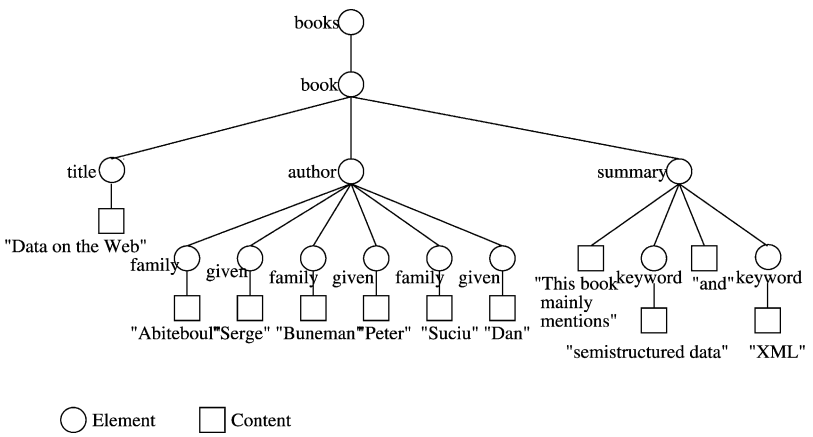


Fig. 2. Tree representation of XML data.

Example 4. ($k = 3$) : Distance ("Data", "Web") ≤ 3. This query means retrieving XML documents in which an occurrence of "Data" is within distance k of "Web"

The hybrid forms of four basic containment queries can be regarded as the core parts of queries over XML IR systems. For example, "/books/book//family/'Abiteboul'" is a hybrid form of direct and indirect containment queries. "/books/book//keyword = 'XML'" is a hybrid form of direct, indirect and tight containment queries.

In this paper, we define the path length of a query as the number of containment relationships in a query. For example, the path length of Example 2 query is four, since there are four containment relationships ("books" and "book", "book" and "summary", "summary" and "keyword", and "keyword" and "XML") in the query.

4. Extending the inverted index

In this section, we show how 2-INDEX approach extends the inverted index for processing containment queries and explain its problems. To solve problems caused by 2-INDEX approach, we, then, suggest a newly extended inverted index.

4.1. 2-INDEX approach and its problems

The inverted index, which is very popular in traditional IR systems, is a technique based on words to make an index for the text to enhance the speed of search activities [3]. The classic inverted index form consists of a text word and its occurrence which enumerates its positions within each document.

2-INDEX approach [46] has two inverted indexes to support containment queries: T-index for indexing text words and E-index for indexing elements. Fig. 3 shows the structure of the two indexes for the XML document in Fig. 1. Each occurrence of a text word is indexed by its document number, its position, and its nesting depth within the document. This is denoted in Fig. 3 as (word,docno,wordno,level). On the other hand, each occurrence of an element is indexed by its document number, its beginning position, its ending position, and its nesting depth within the document. This is denoted in Fig. 3 as (element,docno, begin,end,level). The positions ("begin", "end", and "wordno") in a document are generated by counting word numbers.

The E-index and T-index are mapped into the following two relations (note that primary keys are underlined):

Elements (element, docno, begin, end, level)
Texts (word, docno, wordno, level).

Each of the four containment relationships among elements, attributes, and their contents satisfies the following join conditions. First of all, we assume that the occurrences of an element EL1, an element EL2, a text word W3, and a text word W4 are encoded as (EL1,D1,S1,E1,L1), (EL2,D2,S2,E2,L2), (W3,D3,P3,L3), and (W4,D4,P4,L4), respectively.

- *Indirect Containment*
  (I) EL1 contains EL2 indirectly.
  (1) $D1 = D2$, (2) $S1 < S2$, and (3) $E1 > E2$
  (II) EL1 contains W3 indirectly.
  (1) $D1 = D3$, (2) $S1 < P3$, and (3) $E1 > P3$
- *Direct Containment*
  (I) EL1 contains EL2 directly.
  (1) $D1 = D2$, (2) $S1 < S2$, (3) $E1 > E2$, and (4) $L1 = L2 - 1$
  (II) EL1 contains W3 directly.
  (1) $D1 = D3$, (2) $S1 < P3$, (3) $E1 > P3$, and (4) $L1 = L3 - 1$
- *Tight Containment*
  (I) EL1 contains W3 tightly.
  (1) $D1 = D3$, (2) $S1 = P3 - 1$, and (3) $E1 = P3 + 1$
- *k-Proximity Containment*
  (I) An occurrence of W3 is within distance k of W4
  (1) $D3 = D4$, (2) $P4 - P3 \geq 0$, and (3) $P4 - P3 \leq k$.

Since the general form of containment queries is the mixture of four basic containment queries, containment relationships between elements become the core parts of containment queries. Therefore, it is a very critical issue to find a solution that can process containment relationships between elements effectively. However, 2-INDEX approach incurs a serious problem with respect to performance, since it requires one self-join operation on an "Elements" relation for processing every containment

E-Index(element,docno,begin,end,level)

```
(books,1,1,44,0)(book,1,2,43,1)
(title,1,3,8,2)(author,1,9,28,2)
(family,1,10,12,3)(given,1,13,15,3)
(family,1,16,18,3)(given,1,19,21,3)
(family,1,22,24,3)(given,1,25,27,3)
(summary,1,29,42,2)(keyword,1,34,37,3)
(keyword,1,39,41,3)
```

T-Index(word,docno,wordno,level)

```
(data,1,4,3)(web,1,7,3)
(abiteboul,1,11,4)(serge,1,14,4)
(buneman,1,17,4)(peter,1,20,4)
(suciu,1,23,4)(dan,1,26,4)
(book,1,31,3)(mentions,1,33,3)
(semistructured,1,35,4)(data,1,36,4)
(xml,1,40,4)
```

Fig. 3. E-index and T-index of an XML document in Fig. 1.

/books/book/summary/keyword/'XML'

- **"books" and "book"**
  - Self-join on an "Elements" relation
  - Join conditions:
    $books.docno = book.docno$ and $books.begin < book.begin$ and
    $books.end > book.end$ and $books.level = book.level - 1$
- **"book" and "summary"**
  - Self-join on an "Elements" relation
  - Join conditions:
    $book.docno = summary.docno$ and $book.begin < summary.begin$ and
    $book.end > summary.end$ and $book.level = summary.level - 1$
- **"summary" and "keyword"**
  - Self-join on an "Elements" relation
  - Join conditions:
    $summary.docno = keyword.docno$ and $summary.begin < keyword.begin$ and
    $summary.end > keyword.end$ and $summary.level = summary.level - 1$
- **"keyword" and "XML"**
  - Join between an "Elements" relation and a "Texts" relation
  - Join conditions:
    $keyword.docno = XML.docno$ and $keyword.begin < XML.wordno$ and
    $keyword.end > XML.wordno$ and $keyword.level = XML.level - 1$

Fig. 4. An example of a containment query and its processing by four join operations.

relationship between two elements. The problem is that the number of join operations required is equal to the path length of a query. Fig. 4 shows how many join operations are required for processing "/books/book/summary/keyword/'XML'". As we can see in Fig. 4, it requires four join operations, since the path length of the query is four.

Another serious problem in 2-INDEX approach is the size of relational tables involved in a join operation. If 2-INDEX approach is used for voluminous XML documents, there is always a join operation between two large relations whenever a join operation occurs, since "Elements" and "Texts" relations become huge in general. In Section 6.1, Table 2 shows the size of these two relations when the size of XML documents is 113MB.

4.2. Our approach

In this paper, we extend the inverted index in a different way to solve the problems caused by 2-INDEX approach. As in Fig. 5, there are four inverted indexes. We, then, map four inverted indexes into the following four relations (note that primary keys are underlined):

Path (path, pathID)
PathIndex (pathID, docID, begin, end)
Term (term, termID)
TermIndex (termID, docID, pathID, position).

The relation Path stores data about the paths of elements. That is, it assigns each path to a path identifier

Path (path,pathID)

```
(/books,0)
(/books/book,1)
(/books/book/title,2)
(/books/book/author,3)
(/books/book/author/family,4)
(/books/book/author/given,5)
(/books/book/summary,6)
(/books/book/summary/keyword,7)
```

PathIndex (pathID,docID,begin,end)

```
(0,1,1,44)(1,1,2,43)
(2,1,3,8)(3,1,9,28)
(4,1,10,12)(5,1,13,15)
(4,1,16,18)(5,1,19,21)
(4,1,22,24)(5,1,25,27)
(6,1,29,42)(7,1,34,37)
(7,1,39,41)
```

Term (term,termID)

```
(data,0)(web,1)
(abiteboul,2)(serge,3)
(buneman,4)(peter,5)
(suciu,6)(dan,7)
(book,8)(mentions,9)
(semistuctured,10)(xml,11)
```

TermIndex (termID,docID,pathID,position)

```
(0,1,2,4)(1,1,2,7)
(2,1,4,11)(3,1,5,14)
(4,1,4,17)(5,1,5,20)
(6,1,4,23)(7,1,5,26)
(8,1,6,31)(9,1,6,33)
(10,1,7,35)(0,1,7,36)
(11,1,7,40)
```

Fig. 5. Four inverted indexes to an XML document in Fig. 1.

- **/books/book/summary/keyword**
  (I)After extracting a pathID of "/books/book/summary/keyword" from Path,
  (II)then extract tuples which have the same pathID as it from PathIndex.
- **/books/book//family**
  (I)After extracting pathIDs of "/books/book%/family" from Path,
  (II)then extract tuples which have the same pathID as one of them from PathIndex.

Fig. 6. The processing of containment relationships between elements in RDBMS.

and stores it as a tuple. When an XML document is represented as a tree structure, as shown in Fig. 2, the path of an element is the path on the tree from the root to the node corresponding to the element. For example, the path of "author" element is "/books/book/author" in Fig. 2. The relation PathIndex records the occurrences of each path in the relation Path. Each occurrence of a path is indexed by its identifier, its document number, its beginning position, and its ending position. The position in a document is generated by counting word numbers as 2-INDEX approach.

As we mentioned in Section 4.2, it is a very critical issue to process the containment relationships between elements efficiently. To deal with it, we use the path of an element instead of the name of an element. 2-INDEX approach requires one self-join operation on an "Elements" relation for processing every containment relationship between two elements. Therefore, in cases where a containment query only consists of containment relationships between elements, 2-INDEX approach requires as many join operations as the number of containment relationships between elements. However, our approach requires only one join operation between the relation Path and the relation PathIndex, regardless of the number of a containment relationship between two elements. Fig. 6 shows how containment queries only consisting of containment relationships between elements can be processed by our approach. If a containment query includes "//", every occurrence of "//" is replaced with "%/" by using LIKE predicate in the WHERE clause.

On the other hand, the relation Term stores data about text words. It assigns each text word to an identifier and stores them as a tuple. The relation TermIndex records the occurrences of each text word in the relation Term. Each occurrence of a text word is indexed by its identifier, its document number, its path identifier, and its position in a document.

At this point, it should be noted that we put the path identifier as a field of the TermIndex. By doing so, we can avoid the 2-INDEX approach's problem that the number of join operations increases in proportion to the path length of a containment query. For example, Fig. 7 shows how many join operations are required to process a "/books/book/summary/keyword/'XML'" direct containment query. Our approach only needs two join operations: one join between the relation Term and the relation TermIndex, and one join between the relation Path and the relation TermIndex, while 2-INDEX approach needs four join operations.

2-INDEX approach, as mentioned in Section 4.2, has a drawback that join operations occur between large relations. To solve this, we design four relations instead of two relations in 2-INDEX approach. In Section 6.1, Table 2 shows that the PathIndex and TermIndex relations are relatively smaller than the Elements and Texts relations, respectively, since all of the attributes of the relations are integer types. In Table 2, we can also find that the Path relation is significantly smaller than other relations. The reason is that the size of the Path relation is proportional not to the size of XML documents, but to the size of their Document Type Definition (DTD) documents, which is usually quite smaller than the size of XML documents. One of the main characteristics of XML documents is that a large number of XML documents usually share the same DTD document. In such a case, the size of the Path relation becomes very small as in our case and we therefore can avoid a join between large relations. In the worst case where each XML document has a different DTD document, the Path relation might become large. However, this worst case is very uncommon and we therefore do not consider the case in this paper.

5. Processing containment queries in the RDBMS

In this section, we compare our approach with 2-INDEX approach in processing indirect containment queries and direct containment queries in the RDBMS. Examples of containment queries used in this section are from examples in Section 3. The comparison of our approach and 2-INDEX

/books/book/summary/keyword/'XML'

```
(I)After extracting the pathID, P1 of "/books/book/summary/keyword" from
   the Path relation and the termID, T1 of "XML" from the Term relation,
(II)then, from the TermIndex relation, extract tuples which have the same
    pathID and termID as P1 and T1, respectively.
```

Fig. 7. An example of a containment query and its processing by two join operations.

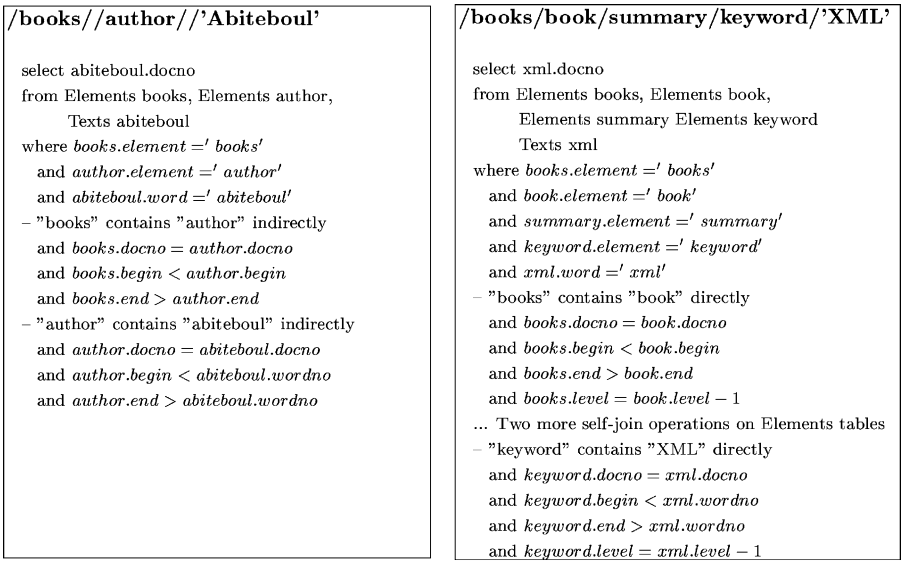


Fig. 8. Processing of indirect and direct containment queries in 2-INDEX approach.

approach for tight containment queries and k-proximity containment queries can be found in Appendix A.

Firstly, we show how to process an indirect containment query. Figs. 8(a) and 9(a) show how to convert "/books//author//'Abiteboul'" into a SQL statement in 2-INDEX approach and our approach, respectively. 2-INDEX method requires two join operations: a self-join on the Elements relation, and a join between the Elements relation and the Texts relation. Our method requires also two join operations: a join between the Path relation and the TermIndex relation, and a join between the Term relation and the TermIndex relation. Although both approaches require two join operations, 2-INDEX approach causes the join operations between larger relations. In addition, in 2-INDEX approach, the number of join operations required is equal to the path length of an indirect containment query, while our approach requires only two join operations, regardless of the path length.

Secondly, in processing a direct containment query, Figs. 8(b) and 9(b) show how to convert "/books/book/summary/keyword/'XML'" into a SQL statement in both approaches. 2-INDEX approach requires four join operations, since there are four direct containment relationships ("books" and "book", "book" and "summary", "summary" and "keyword", and "keyword" and "XML"). However, our approach requires only two join operations: a join between the Path relation and the TermIndex relation, and a join between the Term relation and the TermIndex relation. As in the indirect containment query, our method causes the join operations between smaller relations and requires only two join operations, regardless of the path length.

In cases of direct and indirect containment queries, 2-INDEX approach has no choice but to use nest-loop join, since join conditions have non-equality join predicates. However, our approach can use other join methods, such as hash or sort-merge join, since all join conditions consist of equality join predicates.

6. Experiments

In this section, we show that our RDBMS implementation for processing containment queries is not far behind our IR implementation with respect to performance and

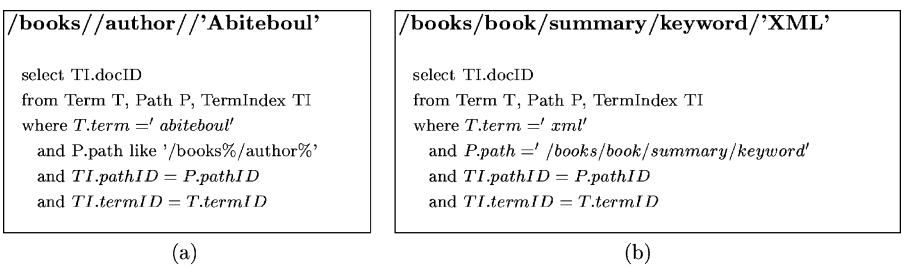


Fig. 9. Processing of indirect and direct containment queries in our approach.

almost always outperforms the RDBMS and IR implementations of 2-INDEX approach.

6.1. Experimental environment

We first explain four methods implemented for comparing performances in this paper.

- 4-Relation method: the method which maps four inverted indexes, as shown in Fig. 5, into four relations (Path, PathIndex, Term, and TermIndex relations) and processes containment queries in the RDBMS.
- 4-B + Tree method: the method which stores the four inverted indexes in the IR engine by using B + -trees and processes containment queries in the IR engine.
- 2-Relation method: the method which maps two inverted indexes, as shown in Fig. 3, into two relations (Elements, and Texts relations) and processes containment queries in the RDBMS.
- 2-B + tree method: the method which stores the two inverted indexes in the IR engine by using B + -trees and processes containment queries in the IR engine (we used Multi-Predicate Merge Join (MPMGJN) [46] for a join operation).

We experimented with Oracle v8.1.7 for 4-Relation method and 2-Relation method. Oracle was run on a 1400 MHZ PIV machine running Microsoft Window 2000 Professional. The main memory size was 768MB and the size of a buffer cache in Oracle was 150MB. We used only the primary keys as RDBMS indexes and JDBC 2.0 for connecting an Oracle server.

On the other hand, 4-B + Tree method and 2-B + Tree method were written in Java and used the BerkeleyDB library [41] to store the inverted indexes. BerkeleyDB is a toolkit that provides B + -tree, Extended Linear Hashing, Queue, and Fixed and Variable-length records as access methods. We used its B + -tree as the access method. The IR engines of 4-B + Tree method and 2-B + Tree method were run on an 1400 MHZ PIV machine running Microsoft Window 2000 Professional. The main memory size was also 768MB.

We used four XML datasets in our experiments. The first was a "Companies" dataset containing the information about various companies in America, the second was a "Cars" dataset containing the information about various automobiles, the third was a dataset of Shakespeare plays, and the fourth was a dataset of DBLP bibliography files. Table 1 shows the sizes of four XML datasets, the Oracle relational table sizes, and the sizes of the inverted indexes stored in BerkeleyDB B + -tree. Table 2 shows the size of each relational table in 4-Relation method and 2-Relation method.

Table 1
Four XML datasets used for experiments

	Companies	Cars	Shakespeare	DBLP
Size of XML documents (MB)	5	19	8	81
Table size in 4-Relation (MB)	190	190	190	190
Inverted index size in 4-B + Tree (MB)	133	133	133	133
Table size in 2-Relation (MB)	215	215	215	215
Inverted index size in 2-B + Tree (MB)	155	155	155	155

Table 2
The size of relational tables

		Size
4-Relation	Term relation	25MB
	TermIndex relation	110MB
	Path relation	26KB
	PathIndex relation	55MB
2-Relation	Texts relation	138MB
	Elements relation	77MB

6.2. Experimental results

We executed three containment queries for each XML dataset. We chose the queries variously from one to five in the path length of the queries. Also, we considered various cases in the frequencies of elements and text words in XML documents. Table 3 shows the raw execution time of the twelve queries on the four implementations. Fig. 10 shows the performance ratios of 2-Relation, 2-B + Tree, and 4-B + Tree to 4-Relation.

Fig. 10(a) shows that 4-Relation method always outperforms 2-Relation method not only in cases (Q2, Q3, Q7, Q8, Q9) in which the path length of a query exceeds two, but also in cases (Q1, Q4, Q5, Q6, Q10, Q11, Q12) in which the path length does not exceed two. We identified two causes of this significant difference in the performance between 4-Relation method and 2-Relation method: (1) 2-Relation

Table 3
The raw execution time (ms) of the 12 queries

XML dataset	Query	4-Relation	4-B + Tree	2-Relation	2-B + Tree
Companies	Q1	90	140	210	106
	Q2	150	30	2533	982
	Q3	250	290	3976	1072
Cars	Q4	220	110	230	80
	Q5	60	20	90	141
	Q6	90	70	120	131
Shakespeare	Q7	60	50	471	50
	Q8	230	150	41,229	3461
	Q9	60	61	14,240	980
DBLP	Q10	40	40	4797	51
	Q11	190	90	26,508	540
	Q12	140	100	5037	153

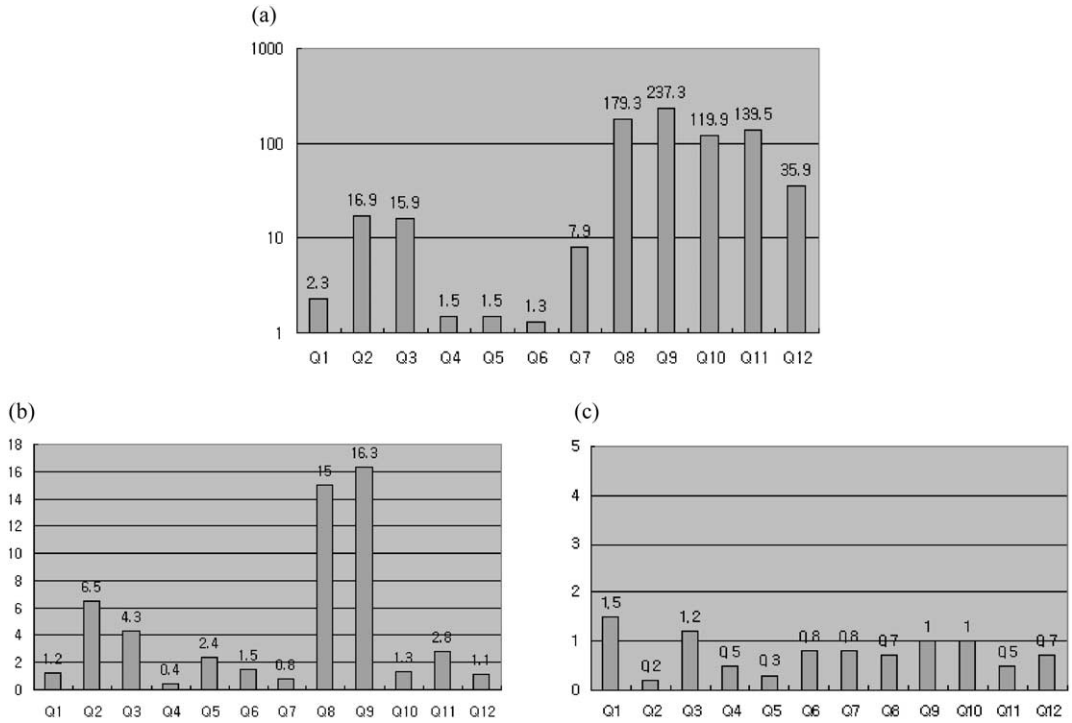


Fig. 10. Performance ratios. (a) 2-Relation/4-Relation performance ratios (log scale). (b) 2-B + Tree/4-Relation performance ratios. (c) 4-B + Tree/4-Relation performance ratios.

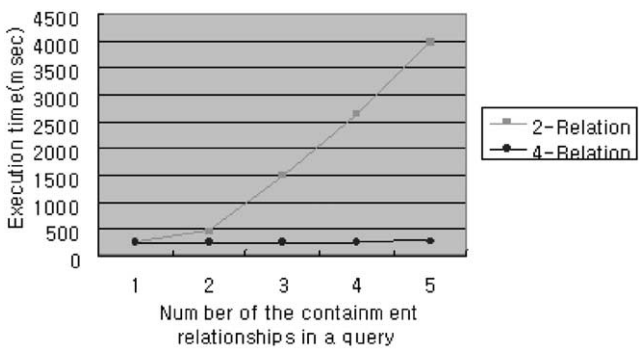
method requires join operations between large relational tables, and (2) it requires as many join operations as the path length of a query.

The characteristic of queries that 2-Relation method is not much worse in terms of performance than 4-Relation method is that the path length of a query does not exceed two and at the same time, the number of tuples participating in join operations is less than tens of thousands. Q1, Q4, Q5, and Q6 belong to such a case. On the other hand, in cases (Q2, Q3, Q7, Q8, Q9) in which the path length of a query exceeds two or cases (Q10, Q11, Q12) in which although the path length does not exceed two, the number of tuples taking part in join operations is greater than tens of thousands, the performance of 2-Relation method lags far behind that of 4-Relation method.

Fig. 10(b) shows that 4-Relation method outperforms 2-B + Tree method in most of the queries. Especially, in cases (Q2, Q3, Q8, Q9) in which the path length of a query exceeds two, the performance ratios of 2-B + Tree method to 4-Relation method are large, while in other cases, the ratios are within three.

As in Ref. [46], the performance of 2-B + Tree method is much better than that of 2-Relation method. Zhang et al. [46] identified two causes for this significant gap of the performance. One is that MPMGJN employed by the IR engine outperforms the standard RDBMS join algorithms, the other is that the RDBMS has much lower cache utilizations than the IR engine. However, Fig. 10(c) shows that the performance ratios between 4-B + Tree method and 4-Relation method are within two, except for Q2 and Q5. The reason is that our RDBMS implementation (4-Relation method) eliminates the gap of the performance between the RDBMS and the IR engine caused by join operations, since the number of join operations does not increase in proportion to the path length of a query and join operations happen between smaller relational tables.

Fig. 11 shows that the execution time of 2-Relation method increases exponentially, as the number of containment relationships in a query (the path length of a query)



1: /companies//'milwalkee'
2: /companies/company//'milwalkee'
3: /companies/company/profile//'milwalkee'
4: /companies/company/profile/address//'milwalkee'
5: /companies/company/profile/address/city='milwalkee'

Fig. 11. Variations of the execution time on the path length of a query.

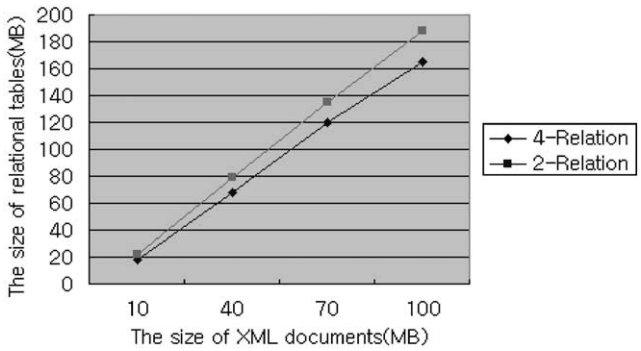


Fig. 12. Relational table size on the size of XML documents.

increases, while there is almost no change in the execution time of 4-Relation method. That is, it shows that 4-Relation method is scalable to the path length of a query.

6.3. Analysis of real space restrictions

Fig. 12 shows that, as the size of XML documents increases, the size of relational tables required for storing the inverted indexes in 4-Relation method increases almost linearly within two times to the size of original XML documents. On the other hand, 4-Relation method requires less storage approximately from 10 to 15% than 2-Relation method.

The main reason why the size of relational tables is larger than the size of XML documents is that we construct the inverted indexes in the granularity of XML components (elements, attributes, and text words) rather than documents. Building the inverted indexes in such a fine granularity is essential for supporting containment queries. However, we are aware that such an increase in the size of the inverted indexes might not be appropriate for some applications. We therefore plan to research efficient database compression techniques and approximate inverted index techniques as our important future work [17].

7. Conclusion

From our experimental results, our IR approach (4-B + Tree method) using B + -trees is almost always the best choice and sometimes two or more times faster than our RDBMS approach (4-Relation method). However, the performance ratios between our RDBMS approach and our IR approach are usually less than two, and further, we showed that our RDBMS implementation significantly outperforms the RDBMS implementation and the IR implementation of 2-INDEX approach. Therefore, our research result suggests that using an RDBMS is not far behind using B + -trees in the performance to store the inverted indexes and process containment queries, and moreover, our approach can be sufficiently used in implementing XML IR systems based on an RDBMS.

In Section 1.2, we mentioned the advantages of using the RDBMS. In the near future, various organizations and companies will adopt XML as their document format and the RDBMS will play a significant role in storing XML documents and processing the queries. However, the shortcoming of our suggested approach is that the size of relational tables is larger than original XML documents. Therefore, we plan to research efficient database compression techniques and approximate inverted index techniques which significantly lessen the size of the inverted indexes and show comparable performance. And as another future work, we will conduct a performance comparison between our approach and the indexing techniques to XML documents internally offered by commercial DBMSs, such as Oracle 9i [21], by using the extensible indexing techniques provided by commercial DBMSs.

Acknowledgements

This work was supported by the Brain Korea 21 Project.

Appendix A. Processing tight and k-proximity containment queries in our approach and 2-INDEX approach

Figs. A1(a) and A2(a) show how to convert "//given = 'Peter'" into a SQL statement in both approaches. 2-INDEX approach requires one join operation, since there is one

//given='Peter'

```
select given.docno
from Elements given, Texts peter,
where given.element =' given'
  and peter.word =' peter'
– "given" contains "Peter" tightly
  and given.docno = peter.docno
  and given.begin = peter.wordno – 1
  and given.end = peter.wordno + 1
```

(a)

Distance("Data","Web") ≤ 3

```
select data.docno
from Texts data, Texts web,
where data.word =' data'
  and web.word =' web'
– "Data" is within distance 3 of "Web"
  and data.docno = web.docno
  and web.wordno – data.wordno ≥ 0
  and web.wordno – data.wordno ≤ 3
```

(b)

Fig. A1. Processing of tight and k-proximity containment queries in 2-INDEX approach.

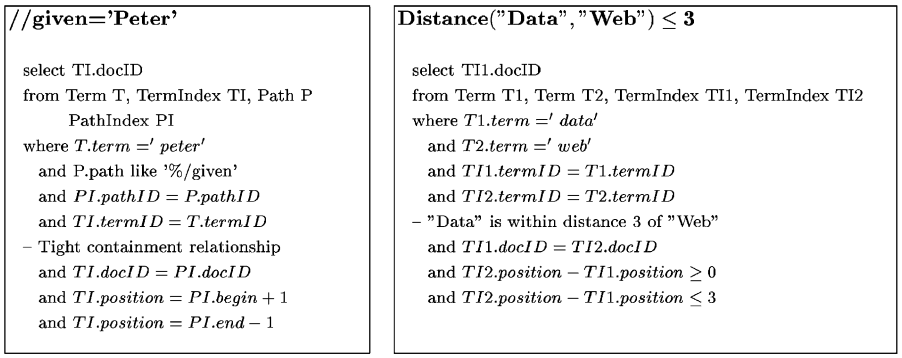


Fig. A2. Processing of tight and k-proximity containment queries in our approach.

containment relationship between a "given" element and a "Peter" text word. On the other hand, our approach requires three join operations (a join between the Term relation and the TermIndex relation, a join between the Path relation and the PathIndex relation, and a join between the TermIndex relation and the PathIndex relation).

Figs. A1(b) and A2(b) show how to convert "Distance("Data","Web")≤3" into a SQL statement in 2-INDEX approach and our approach, respectively. As in the case of a tight containment query, our approach requires three join operations, while 2-INDEX approach requires one join operation.

Although our approach requires more join operations in both a tight containment query and a k-proximity containment query, in case of a hybrid form of basic containment queries (for example, "/books/book/author/family = ' Suciu'"), our approach requires three join operations, regardless of the path length of the query, while 2-INDEX approach requires four join operations between larger relations.

References

[1] T. Arnold-Moore, M. Fuller, B. Lowe, J. Thom, R. Wilkinson, The elf data model and sgql query languages for structured document databases, Proceedings of the Australasian Database Conference (1995).
[2] R. Baeza-Yates, G. Navarro, Integrating contents and structure in text retrieval, SIGMOD Record 25 (1) (1996) 67–69.
[3] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison-Wesley, Reading, MA, 1999.
[4] R. Bayer, XML database: modelling and multidimensional indexing, DEXA (2001).
[5] G. Elizabeth Blake, M.P. Consens, P. Kilpelainen, P.-A. Larson, T. Snider, F.Wm. Tompa, Text/relational database management systems: harmonizing SQL and SGML, Proceedings of the International Conference on Applications of Databases (1994).
[6] T. Bray, J. Paoli, C. Sperberg-McQueen, Extensible markup language (XML) 1.0., Technical report, W3C Recommendation, 1998.
[7] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, M. Stefanescu, XQuery: a query language for XML, Technical report, W3C Working Draft, February 2001.
[8] J. Clark, S. DeRose, XML path language (XPath) Version 1.0., Technical report, W3C Recommendation, November 1999.
[9] C.L.A. Clarke, G.V. Cormack, F.J. Burkowski, Scheme-independent retrieval from heterogeneous structured text, Fourth Annual Symposium on Document Analysis and Information (1995).
[10] C.L.A. Clarke, G.V. Cormack, F.J. Burkoswki, An algebra for structured text search and a framework for its implementation, The Computer Journal 38 (1) (1995) 43–56.
[11] B.F. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, M. Shadmom, A fast index for semistructured data, Proceedings of the Conference on Very Large Data Bases (2001).
[12] T. Dao, R. Sacks-Davis, J.A. Thom, Indexing structured text for queries on containment relationships, Proceedings of the Australasian Database Conference (1996).
[13] S. Dessloch, N.M. Mattos, Integrating SQL databases with content-specific search engines, Proceedings of the Conference on Very Large Data Bases (1997).
[14] A. Deutsch, M. Fernandez, D. Florescu, A.Y. Levy, D. Suciu, XML-QL: a query language for XML, Technical report, W3C, August 1998.
[15] A. Deutsch, M. Fernandez, D. Suciu, Storing semistructured data with STORED, Proceedings of the ACM SIGMOD International Conference on the Management of Data (1999).
[16] T. Fiebig, G. Moerkotte, Evaluating queries on structure with extended access support relations, International Workshop on the Web and Databases (2000).
[17] D. Florescu, D. Kossman, I. Manolescu, Integrating keyword search into XML query processing, Processing of the Ninth International World Wide Web Conference (2000).
[18] D. Florescu, D. Kossmann, Storing and querying XML data using an RDBMS, IEEE Data Engineering Bulletin 22 (3) (1999) 27–34.
[19] International Organization for Standardization, Information processing-text and office systems-standard generalized markup language (SGML), iso/iec 8879, 1986.
[20] R. Goldman, J. Widom, DataGuides: enabling query formulation and optimization in semistructured databases, Proceedings of the Conference on Very Large Data Bases (1997).
[21] http://otn.oracle.com/docs/products/oracle9i, 2001.
[22] http://www.exceloncorp.com, 2001.
[23] http://www.softwareag.com/tamino, 2001.
[24] http://www.xyzfind.com, 2001.
[25] IBM, DB2 text extender, http://www-4.ibm.com/software/data/db2/extenders/text.htm.
[26] D.D. Kha, M. Yoshikawa, S. Uemura, An XML indexing structure with relative region coordinate, IEEE International Conference on Data Engineering (2001).
[27] M. Klettke, H. Meyer, XML and object-relational database systems—enhancing structural mappings based on statistics, International Workshop on the Web and Databases (2000).
[28] Q. Li, B. Moon, Indexing and querying XML data for regular path expressions, Proceedings of the Conference on Very Large Data Bases (2001).

[29] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, Lore: a database management system for semistructured data, SIGMOD Record 26 (3) (1997) 54–66.
[30] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, A. Rajaraman, Indexing semistructured data, Technical report, Stanford University, February 1998.
[31] H. Meuss, Indexed tree matching with complete answer representations, PODDP'98 (1998).
[32] H. Meuss, C. Strohmaier, Improving index structures for structured document retrieval, IRSG'99 (1999).
[33] T. Milo, D. Suciu, Index structures for path expressions, Proceedings of the International Conference on Database Theory (1999).
[34] Oracle, Oracle8i intermedia text reference, release 8.1.5., http://oradoc.photo.net/ora81/DOC/inter.815/a67843/toc.htm.
[35] J. Robie, J. Lapp, D. Schach, XML Query Language (XQL), September 1998, http://www.w3.org/TandS/QL/QL98/pp/xql.html.
[36] R. Sacks-Davis, T. Arnold-Moore, J. Zobel, Database systems for structured documents, Proceedings of the International Symposium on Advanced Database Technologies and Their Integration (ADTI'94) (1994).
[37] J. Schanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, J.F. Naughton, Relational databases for querying XML documents: limitations and opportunities, Proceedings of the Conference on Very Large Data Bases (1999).
[38] J. Schanmugasundaram, E. Schekita, R. Barr, M.J. Carey, B.G. Lindsay, H. Pirahesh, B. Reinwald, Efficiently publishing relational data as XML documents, Proceedings of the Conference on Very Large Data Bases (2000).
[39] A. Schmidt, M.L. Kersten, M. Windhouwer, F. Waas, Efficient relational storage and retrieval of XML documents, International Workshop on the Web and Databases (2000).
[40] T. Shimura, M. Yoshikawa, S. Uemura, Storage and retrieval of XML documents using object-relational database, DEXA (1999).
[41] Sleepycat Software, The Berkeley database, http://www.sleepycat.com.
[42] B. Surjanto, N. Ritter, H. Loeser, XML content management based on object-relational database technology, Proceedings of the International Conference on Web Information Systems Engineering (2000).
[43] R. Whalen, Implementing the full-text search service in sql server, http://msdn.microsoft.com/library/periodic/period00/ewn0092.htm.
[44] T.W. Yan, J. Annevelink, Integrating a structured-text retrieval system with an object-oriented database system, Proceedings of the Conference on Very Large Data Bases (1994).
[45] M. Yoshikawa, T. Amagasa, XRel: a path-based approach to storage and retrieval of XML documents using relational database, ACM Transactions on Internet Technology 1 (1) (2001) 110–141.
[46] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman, On supporting containment queries in relational database management systems, Proceedings of the ACM SIGMOD International Conference on the Management of Data (2001).