관계 DBMS 기반 XML 저장시스템 상에서의 XML 뷰 인덱싱☆

XML View Indexing Using an RDBMS based XML Storage System

박 대 성* 김 영 성** 강 현 철***
Dae-Sung Park Young-Sung Kim Hyunchul Kang

요 약

질의 결과를 캐쉬한 후 후속 질의 처리에 사용하는 것은 중요한 질의 최적화 기법으로서 이에는 실체뷰 기법과 뷰 인덱싱이 있다. 관계 데이터베이스에 대하여 많이 연구된 이들 기법은 XML이 웹 상에서 데이터 교환의 표준으로 부각된 이래 최근 XML 데이터에 대해서도 연구되고 있다. XML 뷰 인덱싱은 XML 질의의 결과인 XML 뷰 xv를 XML 뷰 인덱스(XVI)라는 구조로 표현한다. XVI는 xv의 소스 문서 엘리먼트들의 식별자 및 xv에 대한 정보를 저장한다. xv에 대한 XVI는 XML 엘리먼트의 식별자만을 저장하지 엘리먼트 자체를 저장하는 것이 아니다. 따라서 xv가 요청되면 그것의 XVI를 통해 xv의 하부 소스 XML 문서에 대해 실체화(materialization) 과정이 수행되어야 한다. 본 논문은 XVI 관리 시스템과 관계 DBMS 기반의 XML 저장 시스템의 통합 문제를 다룬다. 제안된 시스템은 두개의 서로 다른 상용 관계 DBMS에 대하여 Windows 2000 Server 환경에서 각각 Java로 구현되어 XML 뷰 인덱싱을 통한 XML 질의 처리의 성능 향상 및 XML 뷰 인덱싱의 오버헤드 평가에 이용되었다. 성능 실험 결과 관계 DBMS 기반의 XML 저장 시스템 상에서 XML 뷰 인덱싱은 매우 효율적이며 그 오버헤드는 미미하다는 것을 확인하였다.

Abstract

Caching query results and reusing them in processing of subsequent queries is an important query optimization technique. Materialized view and view indexing are the representative examples of such a technique. The two schemes had received much attention for relational databases, and have been investigated for XML data since XML emerged as the standard for data exchange on the Web. In XML view indexing, XML view xv which is the result of an XML query is represented as an XML view index (XVI), a structure containing the identifiers of xv's underlying XML elements as well as the information on xv. Since XVI for xv stores just the identifiers of the XML elements not the elements themselves, when xv is requested, its XVI should be materialized against xv's underlying XML documents. In this paper, we address the problem of integrating an XML view index management system with an RDBMS based XML storage system. The proposed system was implemented in Java on Windows 2000 Server with each of two different commercial RDBMSs, and used in evaluating performance improvement through XML view indexing as well as its overheads. The experimental results revealed that XML view indexing was very effective with an RDBMS based XML storage system while its overhead was negligible.

☞ Keyword : XML, View Indexing, Cache Technique, Web based Databases, Database backed Web Application

1. Introduction

Caching the query results and subsequently reusing them for the same or relevant queries is an important query optimization technique, having received much attention in relational database systems [1][2]. For relational query caching, the schemes called materialized view

* 준 회 원 : 중앙대학교 대학원 컴퓨터공학과 석사
dspark@telcoware.com
** 준 회 원 : 중앙대학교 대학원 컴퓨터공학과 박사
yskim@dblab.cse.cau.ac.kr
*** 정 회 원 : 중앙대학교 컴퓨터공학과 교수
hckang@cau.ac.kr
[2004/03/18 투고 - 2004/05/03 1차 - 2004/12/07 2차 - 2005/04/20 심사완료]
☆ 이 논문은 2004년도 중앙대학교 학술연구비(일반연구비) 지원에 의한 것임

or view indexing can be employed. A relational view can be materialized by storing the tuples of the view [1]. In relational view indexing, on the other hand, a view is represented as a view index which consists of the identifiers of the tuples qualified for the view so that the view can be rapidly retrieved by materializing the tuples through their identifiers [15][16]. In retrieving a view with view indexing, it takes time for materializing the view index which is not required with the materialized view scheme. The space overhead of view indexing, however, is much less than that of the materialized view scheme. Both materialized view and view indexing schemes incur the overhead of maintaining consistency of the materialized view or of the view index whenever their underlying source data is updated. In retrieving a view, the materialized view scheme always outperforms view indexing if not considering the overhead of consistency maintenance. The materialized view scheme, however, could incur formidable space overhead especially when a large number of views need to be supported. View indexing trades the extra time it takes for materialization for avoiding space problem and for achieving scalability in terms of the number of views cached. For a view which is not implemented either in view indexing or in materialized view, it is recomputed from scratch every time it is requested. We call such traditional view retrieval scheme as query recomputation.

Both materialized view and view indexing schemes began to be addressed for semi-structured data [3-8] from the late 90s, and also for XML data [9-14] recently. This paper is on XML view indexing. XML view xv which is the result of an XML query is represented as an XML view index(XVI), a structure containing the identifiers of xv's underlying XML elements as well as the information on xv. Since XVI for xv stores just the identifiers of the XML elements not the elements themselves, when xv is requested, its XVI should be materialized against xv's underlying XML documents. An alternative scheme of implementing XML views for query performance is to maintain the views as materialized ones. Also the baseline scheme of supporting XML views is query recomputation.

The core technical components required to realize XML view indexing include (1) a data structure for an XVI, (2) an algorithm to create an XVI when its corresponding XML query is evaluated, (3) an algorithm to materialize an XVI against its underlying XML documents, and (4) an algorithm to incrementally maintain consistency of an XVI given a update against the view's underlying XML documents, where three techniques are needed. First, it needs to check if or not the update is relevant to the view. Secondly, if it is relevant, it needs to generate data for incremental refresh of the XVI. Finally, it needs to install it to the XVI. The data structure of an XVI should be devised in a way to efficiently support all these algorithms.

In [11], these technical issues were addressed and solutions to them were proposed. An XVI management system that operates on top of a DOM-based XML repository was presented. In this paper, we address the problem of integrating the XVI management system of [11] with an RDBMS-based XML stor-

age system. Since the specification of XML 1.0 was released in 1998 by W3C, very active research on XML data management has been conducted. Among others, the problem of storing XML documents in a relational database has received increasing attention because of its pragmatic importance [17-22]. It is expected that more XML data would be stored in the RDBMS in the future. Major RDBMS vendors have already released their versions with XML support. Supporting XML view indexing using an RDBMS-based XML storage system would be a practical solution to high performance and scalable XML query processing, which is crucial for efficient support of XML database-backed Web applications.

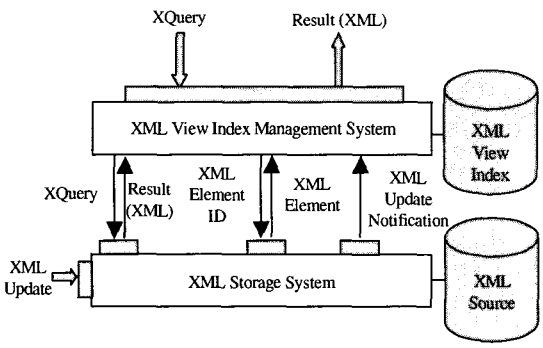
We present an architecture of integrating an XVI management system with an RDBMS-based XML storage system, investigate the technical issues involved in such integration, and propose solutions to them. The proposed system was implemented in Java on Windows 2000 Server with each of two different commercial RDBMSs, and used in evaluating query performance improvement through XML view indexing as well as its overheads.

The rest of this paper is organized as follows. Section 2 surveys the related work. Section 3 presents the architecture of the view indexing system using an RDBMS and deals with its technical issues. Section 4 reports the implementation and the results of performance evaluation. Finally, Section 5 gives some concluding remarks.

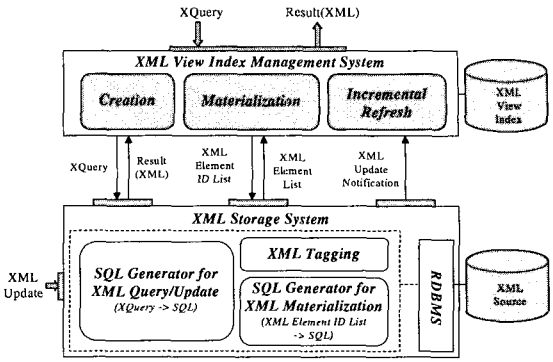
2. Related Work

The query modification scheme [23] was the first proposal for implementing relational views. After that, database snapshots and materialized views along with their incremental refresh had received much attention in the context of relational databases [1]. The idea of representing the query result in an index for fast retrieval of the same query result was also first investigated in the relational database context [15][16]. In [15], the result of a join between two relations R and S is represented as a join index which is a collection of tuple identifier(TID) pairs. Each pair (r,s) points to the joined pair of tuples of R and S. Algorithms that can efficiently process join and semijoin queries through the B+ tree indices on the TIDs in a join index were presented. In [16], the indices for the selection view as well as the join counterpart were investigated. Deferred incremental refresh of these indices against the updates done to the underlying relations were considered. Efficient interleaving of materialization of the query result through these indices and refreshing them for the updates accumulated in the update log was proposed.

As for XML view indexing, it is in infancy yet. In [11], a data structure of XVI and the algorithms to implement the XVI management subsystem of an XML view indexing system were proposed. A subset of FLWR experssion of XQuery was considered for defining XML views, and insertion/deletion of XML elements as well as modification of text values of XML elements were considered as XML updates. A DOM-based XML repository was assumed as the underlying XML storage. An interface between the XVI management system and the underlying XML storage was defined, yielding



〈Figure 1〉 Architecture of XML View Indexing System in [11]



〈Figure 2〉 Architecture of XML View Indexing System Using RDBMS

a framework of interoperability between the two subsystems in an XML view indexing system. In [9][10], a technique to cache an XML view and to maintain it with an index structure was investigated. The XML sources are stored in the binary form of persistent DOMs, and the views are defined in a subset of XQL. The updates considered were insertion/deletion of XML elements and modification of text values of XML elements. An auxiliary information structure called the aggregate path index(APIX) which holds the collection of qualified data objects with respect to the query pattern is used to check the update's relevance to the cached views. The APIX is generated when the view is initially cached and maintained against the subsequent updates on the XML sources.

The approach to XML view indexing in this paper is different from the previous work. It employs the XML storage system based on an RDBMS. Although much work has been done on storing and querying XML data with an RDBMS, there are many issues to be addressed to support XML view indexing with an RDBMS-based XML storage.

3. XML View Indexing Using an RDBMS

In [11], a data structure of the XVI for an XQuery query in FLWR expression and the algorithms to create, incrementally refresh, and materialize an XVI were proposed. The updates to the underlying source XML documents considered were element insertion, deletion, and modification. Figure 1 shows the architecture of the XML view indexing system of [11]. It consists of the XVI management system and the underlying XML storage system, which is a DOM-based XML repository. The former takes XQuery queries and returns their results in XML. In doing so, it may conduct creation, incremental refresh, and materialization of the XVIs. The latter supports the following three functionalities: (1) Given an XQuery query, it returns its result in XML. (2) Given the identifier of an XML element, it returns the corresponding element in XML. (3) When an XML update is processed, it notifies it to the XVI management system. The XVI management system of [11] is generically designed so that it may work on any

type of XML storage system as long as it supports the aforementioned three functionalities.

Figure 2 shows the modules of the XVI management system and those of the XML storage system when it is implemented with an RDBMS. The XVI management system consists of three major modules that carry out the creation, materialization, and incremental refresh of XVIs, respectively, as described in [11]. As for the XML storage system, on top of the RDBMS, there are three major modules that carry out (1) transformation of XML queries and updates to their corresponding SQL counterparts, (2) generation of SQL queries for the list of XML element identifiers in an XVI which are to materialize the XVI, and (3) XML tagging against the SQL result set.

Note that interfacing between the two subsystems in Figure 2 is slightly changed with respect to XVI materialization compared with Figure 1. Now, a list of XML element identifiers is given by the XVI management system, and its corresponding list of XML elements are returned. With an RDBMS-based XML storage, XML data retrieval is conducted by executing SQL statements. As such, if XVI materialization is implemented as a sequence of independent tasks in each of which just one XML element for some element identifier cached in the XVI is retrieved as in Figure 1, too many SQL statements might be invoked, incurring formidable overhead.

In this paper, we employ the exactly same XVI management system described in [11] except for the minor change in the way it materializes an XVI, which is described above and further in Section 3.3. As for the XML storage system based on an RDBMS, to support the interface functions for the XVI management system specified in Figure 2, the technical issues involved in implementing its modules need to be solved. In the rest of this section, we describe those issues and solutions to them.

3.1 Table Schema

The first issue is the design of relational table schema to store XML source documents. There are four tables named Elem_Info, Elem_Content, Elem_Path, and Elem_Materialization used, the former three for storing the XML source documents, and the last one for XVI materialization. In this subsection, we explain the first three tables. The last one is explained in the next subsection.

When an XML document is stored, it is first assigned a unique document identifier (Did) and decomposed into elements so that each element is mapped to relational tuples of the first three tables with a unique identifier (Eid), which is assigned in a monotonically increasing way from the root element to its subelements in the preorder traversal of the XML tree. Note that an XML document is usually modeled as a node-labeled ordered tree, and each XML element, be it leaf or non-leaf, corresponds to a node of that tree. Eid is not just the unique identifier of the element but carries information on the structural information (e.g., ancestor-descendant relationship) and on the order of the elements in a document [24]. Such Eid is very useful in XVI materialization as well as in XML tagging (see the next subsections).

The Elem_Content table stores the text

(PCDATA) of the leaf XML elements. In this paper, for ease of exposition, we consider the data-centric XML documents only where the text can appear only at the leaf elements. Extension to cover the text-centric counterpart as well is straightforward. It consists of columns Did, Eid, and Content where Eid stores the identifier of the leaf element and Content stores its text. The Elem_Path table stores every unique path from the root element to an element in the document that appears at least once along with its unique path identifier assigned. The <path, id> pair is stored in columns Epath and EpathId.

The rest of information on each element is stored in the Elem_Info table, which consists of columns Did, Eid, Ename, EpathId, ParentEid, RmdEid, IrmcEid. Ename stores the tag name of the element. EpathId stores the identifier of the path from the root of the document to that element. ParentEid stores the identifier of the current element's parent element, storing parent-child relationship among the elements. RmdEid(Rightmost descendant Eid) stores the identifier of the current element's rightmost descendant element. RmdEid stores a form of ancestor-descendant relationship among the elements, which is useful when materializing an XML element e given its identifier (i.e. retrieving all the tuples necessary to construct e). RmdEid is also for supporting XML updates along with IrmcEid(Imaginary Rightmost Child Eid). As described above, the Eids assigned in the preorder traversal of the XML tree can represent the structural relationship among the XML elements. The state-of-the-art XML query processing algorithms are centered around such preorder-based XML numbering scheme [24-27]. We also take advantage of such Eid numbering in retrieving the tuples and tagging them to construct an XML subtree in this paper. With insertions and deletions of elements, however, the structural change occurs in the document, and renumbering of Eids might be inevitable. As such, the XML numbering scheme which avoids renumbering altogether against updates is desirable. Such scheme is called update robust. IrmcEid along with RmdEid is for update robust XML numbering, which is beyond the scope of this paper. Its detailed description is in [28].

As for attributes in XML elements, they are treated as elements as in [17]. Storing attributes in a separate table, our table schema looks like the four table-based XML storage first proposed in [20]. But the schema of this paper is different from those in the previous work. It supports XML updates without renumbering Eids as well as efficient query processing, and thus, can efficiently support XML view indexing. The update robust XML numbering scheme, however, could also be embedded in other alternative approaches to XML storage based on an RDBMS proposed in the literature such as the Edge table approach or the Edge/separate value table approach of [17].

3.2 Generation and Processing of SQL Statements

Since the RDBMS is employed as XML stores, XML to relational mapping is also necessary for transforming XML queries (in a FLWR expression of XQuery, in this paper) into SQL counterparts. There are three cases

where SQL statements are generated and processed.

First, when an XQuery query is given and there does not exist its XVI, it is transformed into the SQL expression, which is then executed against the underlying Elem_Info, Elem_Content, and Elem_Path tables. We call this type of query processing without capitalizing on the cache as query recomputation since for a query, its result is recomputed every time it is issued. If the XVI for the query is also to be created, additional SQL statements to retrieve the information necessary to build the data structure of the XVI [11] are generated.

Secondly, when the update to the XML document is requested, an SQL statement against the Elem_Info and the Elem_Path tables is generated and executed to retrieve information on the elements that are to be affected by the update. When the update is notified to the XVI management system, this information is piggybacked. Then, the requested XML update is also transformed into INSERT, DELETE, or UPDATE statements of SQL. In this paper, we considered insertion /deletion of XML elements and modification of text values of XML elements.

Finally, when an XQuery query is given and there exists its XVI, the SQL statements are generated to materialize the XVI. In this case, the Elem_Materialization table is also accessed along with the Elem_Info and the Elem_Content tables. An XVI consists of the information on the corresponding XML view (e.g., XQuery expression defining the XML view) and a list of identifiers of the elements that constitute the view [11]. Materializing an XVI requires retrieving every XML element whose identifier is cached in the XVI. For an element identifier of a leaf element, just a single element is retrieved, whereas for that of a non-leaf element, say e, the whole subtree of the XML document whose root is e is retrieved. The identifiers of all the elements belong to that subtree are greater than or equal to the identifier of e and less than or equal to that of e's rightmost descendant because the identifiers of all the elements are assigned in the preorder traversal as described in Section 3.1. The identifier of e's rightmost descendant is stored in the RmdEid column of the Elem_Info table. Now, we are ready to explain how the XVI materialization is carried out with the SQL statements. Given a list of element identifiers <Eid1, ... , Eidn> in the XVI, the first SQL statement, which is a bulk insert statement, retrieves all the <Eidi, RmdEidi> pairs for each Eidi, i = 1, , n from the Elem_Info table, and inserts them into the Elem_Materialzation table which consists of the Eid and the RmdEid columns. The subsequent SQL statements which join the Elem_Info, the Elem_Content, and the Elem_Materialzation tables retrieves all the tuples necessary to build all the subtrees for XML element identifiers Eid1, ... , Eidn. The Elem_Materialzation table is a temporary workspace employed for XVI materialization. Thus, after the materialization is over, its tuples are deleted altogether.

3.3 XML Tagging

As shown in Figure 2, the underlying XML storage system interfaces with the XVI management system by returning XML fragment

whether it is the result of an XQuery query or a list of XML elements produced for materializing an XVI. As such, the SQL result sets need to be tagged in XML.

Our XML tagging scheme which transforms the tuple streams of the SQL result sets involved into XML with a stack is very effective in the sense that the tuple streams need not be buffered at all. As the tuple streams are produced, each tuple of a stream is fed into the tagging process which consumes it and throws it away before it needs to consume the next one in that stream. When the last tuple of all the streams is processed, generation of the required XML fragment is completed. Such a pipelined XML tagging is possible because (1) the element identifiers within a document are assigned in the preorder traversal as explained in Section 3.1 and (2) the tuple streams are produced as sorted on the document identifier and on the element identifier within a document due to the order by clause in the SQL statements (i.e., ORDER BY Did, Eid). Such ordering of tuples makes it possible for any XML subtree of the XML documents to be generated in a pipelined way as the tuple streams that correspond to the XML elements belong to the subtree are produced. As mentioned in Section 3.1, with any type of random XML update (e.g., element insertion), the identifiers of all the elements in the updated document including the updated ones remain in the preorder without renumbering the element identifiers at all. As such, this pipelined XML tagging is possible even in the presence of XML updates.

In case of XVI materialization, a list of n XML subtrees is generated if a list of n XML element identifiers is given ($n \geq 1$). The above XML tagging process needs a slight extension to handle this case. When the XVI management system receives the returned list of XML subtrees as an XML fragment, it needs to be able to extract each subtree and match it with its root's element identifier. For this, during tagging, every time a subtree is generated, it is accumulated in a hash object with the root's element identifier as the hash key, and the hash object is returned to the XVI management system.

4. Implementation and Performance Evaluation

The XML view indexing system using an RDBMS in Figure 2 was implemented in Java on Windows 2000 Server with each of two different commercial DBMSs. Let us denote the employed RDBMSs as RDBMS A and B. The XVI management system was implemented as described in [11] except just minor adjustment for the new way of XVI materialization described in Section 3.2. As for the underlying XML storage system, it was fully implemented except the function of transforming an XQuery query in FLWR expression to SQL. That function is needed in query recomputation as well as in the creation

〈Table 1〉 The Number of Elements in each XML Source Document Used in Experiments

XML Document Size	Number of XML Elements
10KB	287
40KB	1,145
70KB	1,981
100KB	2,861

```
<root>
 FOR     $U in //USEDCAR,
         $D in $U/DEALER,
         $A in $D/ADDRESS
 WHERE   $A/CITY = "LA"
 RETURN  $D/CAR
</root>
```

〈Figure 3〉 XQuery Query q

of an XVI. In those cases, in the performance experiments, we used the SQL statements manually written. Transformation of XQuery into SQL is well addressed in the literature [29]. Generation of all other SQL statements was implemented.

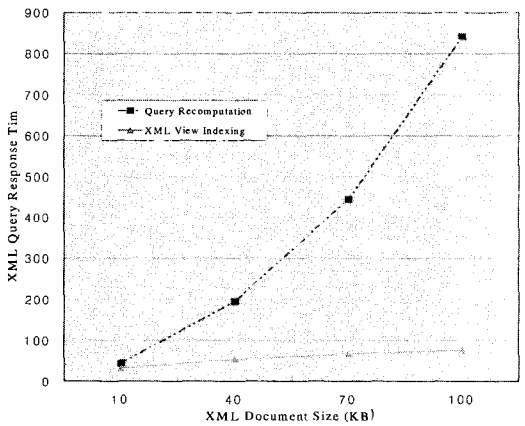
With the implemented system, we conducted a series of experiments to measure the query performance improvement through XML view indexing as well as its overheads. The source XML documents used for the experiments are the ones on the used cars adapted from [30]. We generated four of them whose sizes are 10KB, 40KB, 70KB, and 100KB. Table 1 shows the number of XML elements in each document. Figure 3 shows XML query q in XQuery expression used in the experiments. It is to retrieve the cars which belong to the dealer whose city address is LA. Each car element consists of 6 subelements, one of which is in turn consists of 7 subelements. When q is executed against each of the above four documents, it retrieves about 20% of the source document in terms of the bytes not counting the white space. Each experiment was conducted on two different system implementation and environment. In one implementation, RDBMS A was employed and experiments were done on a system of Pentium III 1.13GHz dual CPU with 2,304 MB memory. In the other implementation, RDBMS B was employed and experiments were done on a system of Pentium Ⅳ

〈Table 2〉 XML Query Response Time (RDBMS A)

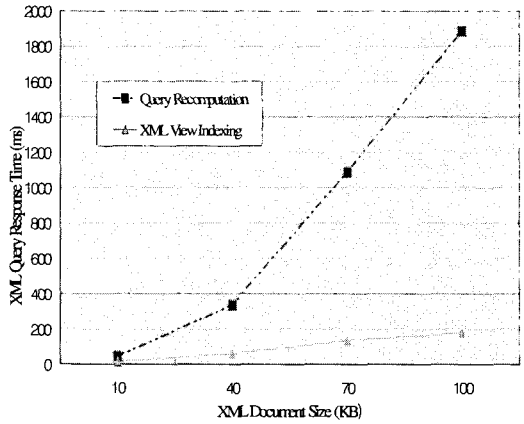
XML Document Size	10KB	40KB	70KB	100KB
Query Recomputation	40ms	195ms	444ms	841ms
XML View Indexing	32ms	54ms	68ms	76ms

〈Table 3〉 XML Query Response Time (RDBMS B)

XML Document Size	10KB	40KB	70KB	100KB
Query Recomputation	46ms	337ms	1088ms	1887ms
XML View Indexing	15ms	62ms	136ms	183ms



〈Figure 4〉 XML Query Response Time (RDBMS A)



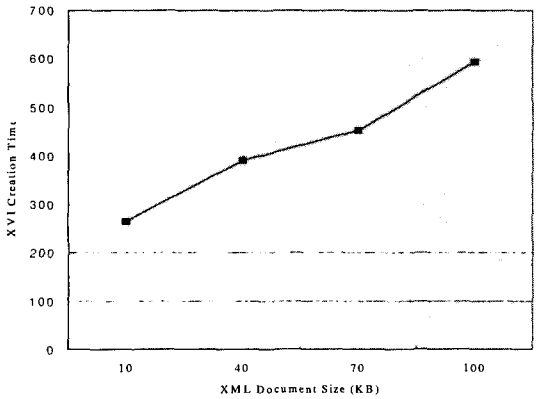
〈Figure 5〉 XML Query Response Time (RDBMS B)

〈Table 4〉 Time for Creating XML View Index (RDBMS A)

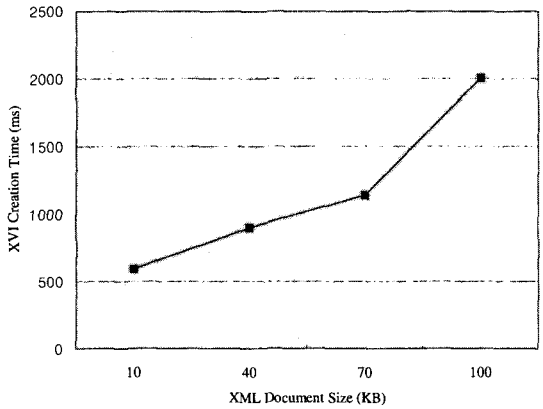
XML Document Size	Time for Creating XVI
10KB	265ms
40KB	391ms
70KB	453ms
100KB	593ms

〈Table 5〉 Time for Creating XML View Index (RDBMS B)

XML Document Size	Time for Creating XVI
10KB	591ms
40KB	901ms
70KB	1142ms
100KB	2003ms



〈Figure 6〉 Time for Creating XML View Index (RDBMS A)



〈Figure 7〉 Time for Creating XML View Index (RDBMS B)

1.80GHz single CPU with 256 MB memory.

Comparing XML view indexing, XML materialized view, and query recomputation schemes, the materialized view scheme outperforms the rest in terms of query response time through a view. Since the response time with materialized view is virtually zero (the query result or the view is always ready to be returned), it shows the optimal response time. The view indexing scheme takes longer because it needs the process of materialization of the view through the XVI. Table 2/Figure 4 and Table 3/Figure 5 show the response times of q measured with query recomputation and with XML view indexing using RDBMS A and B, respectively. They were measured by averaging the results of 20 runs. It is revealed that as the size of the source XML document increases, XML view indexing absolutely outperforms query recomputation.

Meanwhile, the disadvantage of XML view indexing is that it incurs some overheads that are not incurred when XML view indexing is not supported and thus the view is retrieved

〈Table 6〉 Time for XML Element Updates (RDBMS A)

	Without XVI	With XVI
Insertion	312ms	375ms
Modification	63ms	79ms

〈Table 7〉 Time for XML Element Updates (RDBMS B)

	Without XVI	With XVI
Insertion	691ms	721ms
Modification	40ms	60ms

by query recomputation. Such overheads include (i) the time of creating an XVI and (ii) the additional time spent to complete the processing of updates to the XML source. Table 4/Figure 6 and Table 5/Figure 7 show the time it took for creating the XVI of q using RDBMS A and B, respectively. As the size of the source document increases, it took more time. For the small source document (e.g., 10KB), this overhead is huge compared with the time it would take to recompute the view which corresponds to the created XVI (refer to Table 2/Figure 4 and Table 3/Figure 5.) For the large source, however, this overhead is negligible. Besides, note that this overhead is paid just once.

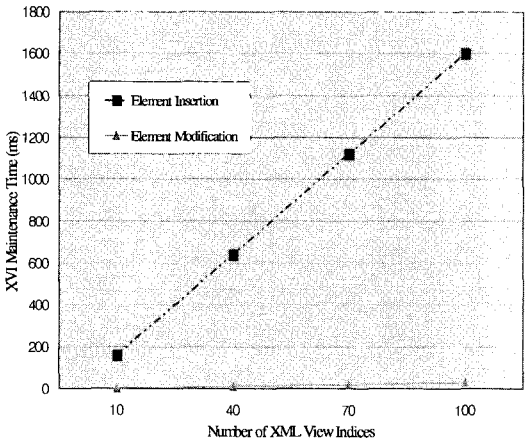
Another overhead in XML view indexing is incurred when XML updates are processed. When the XVIs are maintained, the additional tasks to incrementally refresh them against the source updates are required. For the source document of 100KB, we considered two types of XML updates. One is a simple update to modify the city address (PCDATA) of a dealer. The other is a heavy update to insert a car element. Table 6 shows the times it took to commit these updates when the XVI of q exists (i.e., with XVI) and when it does not (i.e., without XVI), respectively using RDBMS A. Table 7 is for RDBMS B. The difference amounts to the time it took for maintaining consistency of q's XVI. Compared with the time it would take to recompute the view which corresponds to the maintained XVI (refer to Table 2/Figure 4 and Table 3/Figure 5), however, such overhead is

〈Table 8〉 Overhead of Maintaining XML View Indices (RDBMS A)

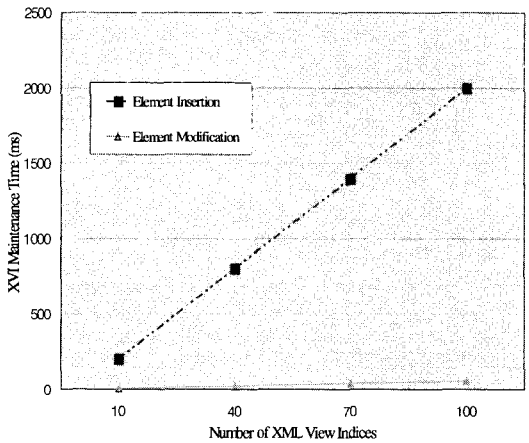
Number of XVIs	10	40	70	100
Element Insertion	160ms	640ms	1120ms	1600ms
Element Modification	3ms	12ms	21ms	30ms

〈Table 9〉 Overhead of Maintaining XML View Indices (RDBMS B)

Number of XVIs	10	40	70	100
Element Insertion	200ms	800ms	1400ms	2000ms
Element Modification	6ms	24ms	42ms	60ms



〈Figure 8〉 Time for Maintaining XML View Indices (RDBMS A)



〈Figure 9〉 Time for Maintaining XML View Indices (RDBMS B)

negligible. Table 6 and Table 7 show the update overhead for one XVI. To see how scalable our XML view indexing is, we estimated the update overhead with respect to the increasing number of XVIs maintained. When the numbers of the XVIs are 10, 40, 70, and 100 (the size of the source document is 100KB, and all the XVIs are the same as q's in this estimation), the times it would take for maintenance of all these XVIs when RDBMS A and B are used are shown in Table 8/Figure 8 and Table 9/Figure 9, respectively. They were estimated by multiplying the number of XVIs maintained to the time of maintaining one XVI measured in Table 6 and Table 7. It turned out that our XML view indexing would be very scalable for the simple updates (i.e., element modification) but would not be so much as that for complicated updates (i.e., element insertion).

Finally, let us compare XML view indexing with XML materialized view scheme. As mentioned above, the former cannot compete with the latter in terms of query response time because the latter achieves the optimal performance. When it comes to the space overhead, however, the latter requires much more space than the former. Table 10 and Figure 10 show the space required to store the XVI of q and the materialized view of q, respectively. Since they are stored outside the RDBMS, the space requirement is independent of the underlying RDBMS used. As expected, as the size of the source document increases, XML view indexing required much less space compared with the materialized view scheme, lending itself to more scalable solution to XML query caching.

5. Concluding Remarks

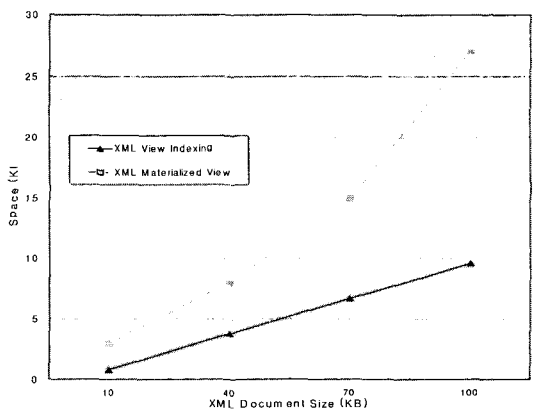
In this paper, we investigated XMLview indexing using an RDBMS-based XML storage system. XML storage using the widely deployed RDBMS is very practical approach, and thus, has received increasing attention. As such, integrating an XML view index management system with an RDBMS-based XML storage system is an interesting challenge.

We presented an architecture of the XML view indexing system using an RDBMS, and dealt with the technical issues involved such as table schema, SQL generation, and XML tagging which are to support XML view index management tasks such as creation, incremental refresh, and materialization of the XML view indices.

The proposed system was implemented with

〈Table 10〉 Space Overhead

XML Document Size	10KB	40KB	70KB	100KB
Size of XVI	0.8KB	3.8KB	6.7KB	9.6KB
Size of XML Materialized View	3KB	8KB	15KB	27KB



〈Figure 10〉 Space Overhead

each of two different commercial RDBMSs, and the query performance improvement through XML view indexing as well as its overheads were measured. It turned out that XML view indexing firmly outperformed query recomputation while its overheads were marginal enough to be compensated with query performance improvement. In the implementation, we employed two different commercial RDBMSs anonymously denoted as A and B, and we could observe that the corresponding experiments show the same pattern of performance results for A and B. This implies that the performance of the XML view indexing system using an RDBMS with JDBC connection which is one of the standard database APIs proposed in this paper is independent of the particular RDBMS employed.

The data structure of XVI is closely interrelated with efficiency of the algorithms for incremental refresh and materialization of XVI. We are currently investigating implementation alternatives to the data structure of XVI and the related algorithms used in this paper in XVI management in order to figure out the performance characteristics and tradeoffs among them and also their suitability for the RDBMS-based XML storage.

References

[1] A. Gupta and I. Mumick, "Materialized Views: Techniques, Implementations, and Applications," 1999, MIT Press.

[2] A. Levy et al., "Answering Queries Using Views," Proc. of ACM Int'l Symp. on PODS, 1995.

[3] S. Abiteboul et al., "Incremental Maintenance for Materialized Views over Semistructured Data," Proc. Int'l Conf. on VLDB, 1998, pp. 38-49.

[4] D. Suciu, "Query Decomposition and View Maintenance for Query Languages for Unstructured Data," Proc. Int'l Conf. on VLDB, 1996, pp. 227-238.

[5] Y. Zhuge and H. Garcia-Molina, "Graph Structured Views and Their Incremental Maintenance," Proc. Int'l Conf. on Data Engineering, 1998, pp. 116-125.

[6] D. Calvanese et al., "Answering Regular Path Queries Using Views," Proc. Int'l Conf. on Data Eng., pp. 389-398, 2000.

[7] D. Florescu et al., "Query Containment for Conjunctive Queries with Regular Expressions," Proc. Int'l Symp. on PODS, 1998, pp. 139-148.

[8] Y. Papakonstantinou and V. Vassalos, "Query Rewriting for Semistructured Data," SIGMODProc. Int'lConf. on Management of Data, pp. 455-466, 1999.

[9] L. Chen and E. Rundensteiner, "Aggregate Path Index for Incremental Web View Maintenance," Proc. 2nd Int'l Workshop on Advanced Issues of E-Commerce and Web-based Information Systems, 2000.

[10] L. Quan et al., "Argos: Efficient Refresh in an XQL-Based Web Caching System," Proc. Workshop on the Web and Databases, 2000, pp. 23-28.

[11] Y. Kim and H. Kang, "XML View Indexing," J. of KISS: Databases, Vol. 30, No. 3, 2003. 6, pp. 252-272.

[12] V. Hristidis and M. Petropoulos, "Semantic Caching of XML Databases," Proc. Workshop on the Web and Databases, 2002.

[13] L. Chen and E. Rundensteiner, "ACE-XQ: A CachE-aware XQuery Answering System," Proc. Workshopon the Web and Databases, 2002.

[14] P. Marron and G. Lausen, "Efficient Cache Answerability for XPath Queries," Proc. the 2nd Int'l Workshop on Data Integration over the Web(DIWeb), 2002, pp. 35-45.

[15] P. Valduriez, "Join Indices," ACM Trans. on Database Systems, Vol. 12, No. 2, Jun. 1987, pp. 218-246.

[16] N. Roussopoulos, "An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis," ACM Trans. on Database Systems, Vol. 16, No. 3, Sep. 1991, pp. 535-563.

[17] D. Florescu and D. Kossmann, "Storing and Querying XML Data Using an RDBMS," IEEE Data Engg. Bulletin, Sep. 1999, pp. 27-34.

[18] A. Deutsch et al., "Storing Semistructured Data with STORED," Proc. ACM SIGMOD Int'l Conf. on Management of Data, 1999, pp. 431-442.

[19] J. Shanmugasundaram et al., "Relational Databases for Querying XML Documents: Limitations and Opportunities," Proc. Int'l Conf. on VLDB, 1999, pp. 302-314.

[20] T. Shimura et al., "Storage and Retrieval of XML Documents Using Object-Relational Databases," Proc. DEXA, 1999.

[21] F. Tian et al., "The Design and Performance Evaluation of Alternative XML Storage Strategies," ACM SIGMOD Record, Vol. 31, No. 1, Mar. 2002, pp. 5-10.

[22] P. Bohannon et al., "From XML Schema to Relations: A Cost-based Approach to XML Storage," Proc. Int'l Conf. on Data Engineering, 2002.

[23] M. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," Proc. ACM SIGMOD Int'l Conf. on Management of Data, 1975, pp. 65-78.

[24] C. Zhang et al., "On Supporting Containment Queries in Relational Database Management Systems," Proc. ACM SIGMOD Int'l Conf. on Management of Data, 2001, pp. 425-436.

[25] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," Proc. Int'l Conf. on VLDB, 2001.

[26] S. Al-Khalifa et al., "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," Proc. Int'l Conf. on Data Engineering, 2002, pp. 141-152.

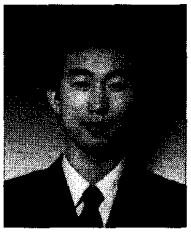
[27] N. Bruno et al., "Holistic Twig Joins: Optimal XML Pattern Matching," Proc. ACM SIGMOD Int'l Conf. on Management of Data, 2002, pp. 310-321.

[28] J. Park and H. Kang, "Handling Updates for Cache-Answerability of XML Queries on the Web," Lecture Notes in Computer Science, Vol. 3112, Jul. 2004, Springer-Verlag, pp. 124-135.

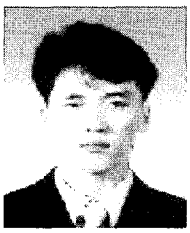
[29] D. DeHaan et al., "A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding," Proc. ACM SIGMOD Int'l Conf. on Management of Data, 2003, pp. 623-634.

[30] D. Maier, "Database Desiderata for an XML Query Language," position paper, QL-98, http://www.w3.org/TandS/QL/QL98 /pp.html.

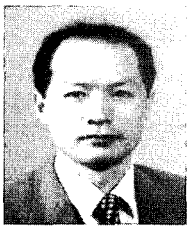
◐ 저 자 소 개 ◑



박 대 성 (Dae-Sung Park)
2003년 중앙대학교 컴퓨터공학과 졸업(학사)
2005년 중앙대학교 대학원 컴퓨터공학과 졸업(석사)
관심분야 : XML 데이터베이스, 음성핵심망
E-mail : dspark@telcoware.com



김 영 성 (Young-Sung Kim)
1997년 중앙대학교 컴퓨터공학과 졸업(학사)
1999년 중앙대학교 대학원 컴퓨터공학과 졸업(석사)
2001년 ~ 2004년 (주)케이포엠, 케이컴스(주) 근무
1999년 ~ 현재 중앙대학교 대학원 컴퓨터공학과 박사과정 재학중
관심분야 : XML 데이터베이스, 웹 데이터베이스, DBMS 엔진
E-mail : yskim@dblab.cse.cau.ac.kr



강 현 철 (Hyunchul Kang)
1983년 서울대학교 컴퓨터공학과 졸업(학사)
1985년 U. of Maryland at College Park, Computer Science(M.S.)
1987년 U. of Maryland at College Park, Computer Science(Ph.D.)
1988년 ~ 현재 중앙대학교 컴퓨터공학과 교수
관심분야 : XML 데이터베이스, 웹 데이터베이스, 이동 데이타베이스
E-mail : hckang@cau.ac.kr