Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Commerce Research and Applications 5 (2006) 66–77

Electronic Commerce Research and Applications

www.elsevier.com/locate/ecra

Efficient indexing technique for XML-based electronic product catalogs

Arzucan Özgür, Taflan İ. Gündem *

Computer Engineering Department, Boğaziçi University, 34342 Bebek, İstanbul, Turkey

Received 22 April 2003; received in revised form 12 February 2004; accepted 10 October 2005
Available online 2 November 2005

Abstract

Electronic product catalogs are considered as one of the main components of e-commerce applications. Efficient processing of queries on product catalogs is important for customer satisfaction. In this paper, we present an indexing structure for processing queries efficiently on natively stored XML-based electronic product catalogs. We also present the performance comparison of our index structure with two alternative approaches. The first is the extended inverted index technique, used in information retrieval and also in product catalogs. The second is the traditional way of traversing the entire XML document for query processing via SAX. The research done in the literature on efficient storage structure for XML is on generic XML documents. In this paper, we focus on efficient index structures for XML-based product catalogs considering their specific needs and properties.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Electronic product catalogs; Semi-structured XML document; Indexing; Efficient query processing

1. Introduction

The Extensible Markup Language (XML) is becoming a standard for Internet data representation as a mark-up language and for data exchange over the Internet [1]. XML is also used for storing semi-structured data such as product catalogs [2]. XML allows users to ask very powerful queries on the web. For example, doing a keyword search of "university", "America", "biology" will probably return thousands of documents most of which are irrelevant. However, asking a query such as "Find the universities in America that offer a biology degree and have an annual fee below $20,000." using an XML based query language such as XML-QL [3], XQuery [4], XQL [5] or LOREL [6] allows the user to get relevant results.

Product catalog is one of the most important components of an e-commerce business as it is the main link between the customer and the supplier. Here the customer stands for the web user, who searches in the product catalog for specific products and has a tendency to purchase them. Supplier stands for the organizations, who sell their products on the web. The current trend is to store the product catalogs in relational database management systems and allow keyword-based search. However due to XML's current popularity, we expect XML to replace at least partially the relational databases for product catalogue representation in the near future. XML can easily represent semi-structured data present in product catalogs and allows precise and powerful queries to be posed over semi-structured data. Also if the product catalogue is stored in XML, there is no need to a transformation of the information in the product catalogue in data exchange, since XML is fast becoming a standard in data exchange. Thus, in this paper, we propose to store product catalogs in the form of XML documents which we think will be the trend in the future.

In order to increase customer satisfaction, products or product groups should be accessible in an efficient way. The proper choice of the physical storage structure of an XML document and the indexing technique used are crucial for efficient query processing. A lot of research has been done to design efficient storage structures and indexing techniques for XML-based documents. In the literature, the index and storage structures proposed such as in [2,7–16] are for generic XML documents and they do not

* Corresponding author. Tel.: +90 212 3596605; fax: +90 212 2872461.
E-mail address: gundem@boun.edu.tr (T.İ. Gündem).

1567-4223/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.elerap.2005.10.007

consider the semantics and the special properties of product catalogs. In this paper, considering the peculiarities of product catalogs and the queries most frequently posed to them, we designed an efficient indexing technique for XML-based electronic product catalogs stored in their original format.

As discussed in [2] there are mainly three approaches for storing XML documents. One approach is to store XML documents in a relational or object-oriented database system. In [7–9] it is examined how to map and store XML documents in a relational database system. A version of Lore [10] examined the use of an object-oriented database system O2 [11] for storing semi-structured data. It is stated in [2] that storing and accessing XML documents via an SQL interface poses an overhead not related to storage. Thus using an object manager such as Shore [12] to store XML data is proposed as an alternative approach. However, object managers do not offer the portability supplied by Relational DBMS [2]. The third and perhaps one of the most frequently used approaches is to store each XML document in a text file. In this approach when an XML query is evaluated against the document, the document is read and parsed into a tree such as a DOM tree first and then this structure is navigated to get the results of the query. The main advantages of this approach can be summarized as follows. It is easy to implement and there is no need for an underlying database or object manager system. However, in this approach the XML file has to be parsed each time a query is evaluated. Also the parsed file must be memory resident during query processing. Thus using DOM is not suitable for disk-resident large product catalogs. As a solution external indices are used.

Storing XML-based product catalogs in relational or in object-oriented database system or using an object manager requires extra work in mapping XML data to relational or object data models. In addition when a portion of the document or the whole document is required, which is frequently the case for product catalogs when they are exchanged or browsed, there is the extra time and effort for reconstruction and loading. Therefore, in this paper, we suggest to store XML-based product catalogs in their original format and propose an indexing structure for this storage.

A lot of research has been done on indexing techniques for efficient query processing of natively stored XML documents in the literature. Some important examples are DataGuides [13], the indexing technique used in the Lore system [10], 1-Indexes and 2-Indexes [14], and the widely used Inverted Indexing technique in information retrieval [15,16]. DataGuides store each path in the XML document starting from the root. They increase the performance of queries that involve navigation of the document from the root by reducing the portion of the document to be scanned. However, DataGuides do not provide any information about the parent–child relationship among elements and attributes of the document. Thus they cannot be used for path queries that do not start from the root. The same problem is also faced by 1-Index. The Lore system proposes a solution to this problem by two additional indexes, Bindex and Lindex [10]. However, since the overall path structure is not stored as in DataGuides, the forward and backward navigation over large datasets causes extensive look-ups and joins. The Inverted Indexing technique [15], which is very popular in traditional information retrieval systems to speed search operations, is extended for XML documents in [16]. This Extended Inverted Indexing technique preserves parent–child and ancestor–descendant relationships among elements and attributes of the XML document and also supports the processing of path queries starting from any arbitrary node. Among the general techniques to index XML documents, this technique seems to be suitable for XML-based product catalogs. Thus it has been used to enhance query processing over XML-based product catalogs in the Agent-Based Electronic Commerce System (ABECOS) Project [17].

In this paper, first an overview of product catalogs in electronic commerce is given. The advantages of storing and presenting catalogs in XML format and the main problem of standardization are discussed in Section 2. The most common query types asked over product catalogs are discussed in Section 3. In Section 4, the Extended Inverted Indexing technique [16] is described. The proposed indexing technique is examined in Section 5. In Section 6, we present the performance comparison of the proposed indexing technique with the traditional system as a baseline, which does not use indices but traverses the entire XML document sequentially for query processing and with the Extended Inverted Indexing technique proposed in [16] and used in [17] for product catalogs. Finally, we discuss the results and conclude in Section 7.

2. Product catalogs in e-commerce

Despite the huge growth of the e-commerce industry and the success of the new businesses such as Amazon, Yahoo or eBay, business-to-consumer e-commerce area has many problems and requires research and improvements especially from the database community.

An electronic commerce application such as a virtual store, typically involves several types of actors such as customers and suppliers. In addition, it also involves a significant amount of data such as product information stored in product catalogs, customer information, order and purchase information, payment and credit card information, delivery information and invoice information. In this paper, we concentrate on electronic product catalogs, which are usually the most significant link between the customer and the supplier and therefore are very important for customer satisfaction. That's why the storage, presentation, design and organization of catalog information are critically important to create brand recognition [18].

In this paper, we view the product catalogs from the point of view of databases. To achieve customer satisfaction the efficiency of query processing over the product catalogs is very important. We suggest storing product catalogs in XML format and propose an efficient indexing structure to increase the efficiency of query processing. In this section the main advantages and drawbacks of storing and retrieving electronic product catalogs in XML format are discussed briefly.

Storing product catalogs in XML format provides rich product content, product classification, simplicity, flexibility, query optimization and interoperability. These are the main requirements a good product catalog should possess [18,19]. In the following paragraphs, we discuss how XML meets these requirements.

Product or service information stored in a product catalog may be in the format of text, images, videos, and PDF's among others. A usable catalog should include rich structured product content. Product content must include detailed product specifications (attributes) that may apply to only a subset of the products, in addition to fields of information common to all the products in the catalog such as product ID and price [18]. These properties make the content of product catalogs semi-structured data. The most suitable form of representing and storing semi-structured data is in the form of XML documents.

The products should be organized and classified into an arbitrary hierarchy of categories and subcategories, which may contain any number of levels [18]. This hierarchy leads to parent–child relationships, which may be perfectly represented by XML format. In addition XML allows the definition of new tags, which can identify the type of each data field stored in the catalog [19]. In this way the product properties are described in a more comprehensive and flexible way.

Another advantage of using XML for storing and presenting product catalogs is its simplicity and flexibility. XML stores information not in a binary format, but in a character-based format. This feature allows the information to be read or modified by any text editor without any need for format conversion [19]. XML allows the information of the catalog to be separated from the presentation. The catalog may be presented using different layouts and formats without modifying the original file where the information is stored [19].

To achieve customer satisfaction, the customer should be able to locate individual products or groups of products quickly and efficiently. This is why the lookup queries over the product catalogs should be executed accurately and efficiently. Since a specific tag identifies each data field in an XML-based catalog, the client application can use the information in the tags when formulating queries. In this way, more relevant results are obtained when compared to keyword-based lookup operations. This is how XML allows query optimization over product catalogs.

Since customers may use different hardware and software platforms, product catalogs should be interoperable with any platform. This requirement is also supported by XML [18,19].

However the main drawback of storing electronic product catalogs in XML format is the lack of standardization. XML is an extensible language and each supplier is free to define his own tags. There is no approved standard that agrees on common tags with common semantics among XML-based product catalogs yet. However, a lot of effort is being spent to find a solution to this problem. Two of the most important ones are RosettaNet [20] and eCo Framework [21].

3. Common query types in product catalogs

There are mainly two types of accesses executed over electronic product catalogs. The first type of access is update such as deletion, insertion or modification. Updates are executed by the vendor or creator (administrator) of the product catalog. The vendor may decide to insert or delete a category type or a product. The vendor may also want to modify a property of a product or category type. However update queries are rare. As stated in [22], the frequency of update queries is approximately 10% of the whole queries. The remaining 90% of queries fall into the second category. The second category of accesses is lookup operations. The efficient processing of lookup queries is very important for customer satisfaction. The customer should be able to locate individual products or groups of products quickly and efficiently. The customer should be able to find products by submitting any property or properties of the searched product. As a result of the lookup query, the complete information requested about a specific product or products should be returned.

We classify the main lookup query types executed over product catalogs into four types. The queries are expressed in XPath standard [23]:

- **Type 1:** In Type 1 queries the customer wants to retrieve the products that have specific properties. Some examples of this type of queries are given below:
  Q1: /catalog/category[@name = 'books']/product[name = 'The Brothers Karamazov']
  This query returns all the products in the 'books' category that have name 'The Brothers Karamazov'.
  Q2: /catalog//product[price[@currency = '$'] < 15]
  This query returns all the products in the catalog whose prices are less than $15.
  Q3: /catalog/category[@name = 'books']/product[name = 'The Brothers Karamazov' and author = 'Fyodor Dostoyevsky']
  This query returns all the products in the product catalog, which are in the 'books' category that have name 'The Brothers Karamazov' and author 'Fyodor Dostoyevsky'.
- **Type 2:** In Type 2 queries the customer wants to retrieve all of the values of a property of a certain type of product. Some examples of this type of queries are given below:
  Q4: //category[@name = 'books']//author
  This query retrieves the list of authors whose books are present in the catalog.

Q5: /catalog/category[@name = 'books']/product/name
This query retrieves the list of the names of the books in the product catalog.

- **Type 3:** Type 3 queries are very common in product catalogs. In this type of queries the customer wants to retrieve all the products in a specific category in the product catalog. Some examples of Type 3 queries are given below:
  Q6: /catalog/category[@name = 'books']
  This query lists all the products in the 'books' category.
  Q7: /catalog/category[@name = 'CDs']
  This query lists all the products in the 'CDs' category.
- **Type 4:** In this type of query, the customer gives a specific property or properties of a product and wants to retrieve some other property or properties of the product. Some examples of Type 4 queries are given below:
  Q8: /catalog/category[@name = 'books']/product[name = 'The Brothers Karamazov']/price
  In this query the price of the book with name 'The Brothers Karamazov' is asked.
  Q9: /catalog/category[@name = 'widgets']/product[name = 'umbrella']/description. In this query the description of the product in the 'widgets' category with name 'umbrella' is asked.

4. Extended inverted indexing technique

Inverted indexing technique [15] has been used in IR (information retrieval) systems widely to speed search operations. It is based on indexing text words and their positions in the documents. Inverted indexing technique has been extended in [16] for XML documents.

This technique preserves the parent–child and ancestor–descendant relationship among elements and attributes. It allows processing of path queries starting from any arbitrary node. Inverted indexing technique has been used widely in IR systems. This extended version seems to be the most suitable technique to index XML-based product catalogs among the general purpose indexing techniques for XML documents discussed in the introduction. We are not aware of any other indexing technique used in indexing XML based product catalogs. This technique is used to enhance query processing over XML-based product catalogs in Agent-Based Electronic Commerce System (ABECOS) Project [17]. Therefore, we have chosen to compare this technique with our proposed method in the performance study.

4.1. The structure of the extended inverted indexing technique

The extended inverted index is composed of a suite of indexes namely *inverted tag index* and *inverted term index*. The former is for indexing the elements and attributes in the XML document and the letter is for indexing the values of them. In [16], term is defined to be each text word in the document. However, we define it here as the value of each element or attribute to make it comparable with our system. Another modification we adapt is to remove the document ID field from the indexes, since we assume that a product catalog consists of a single XML document on which the queries are evaluated. These modifications increase the performance of the extended inverted index for product catalogs.

In Table 1, we display the inverted tag index for portion of the sample product catalog given in Appendix A. The *element* field stores the name of elements and attributes for each occurrence in the document. *Start_offset* and *end_offset* is the beginning and ending position of that element or attribute in the document. For ease of implementation, we take begin and end line of elements and attributes as their start and end offsets (byte offsets can also be used). The *level* field indicates the nesting depth of that element or attribute in the document. For instance the nesting depth of the root element "catalog" is 0 and of "category" element is 1 for the sample product catalog in Appendix A.

Table 2, displays the inverted term index for a portion of the sample product catalog in Appendix A. In this index, each occurrence of an element or attribute is indexed by its value in the *term* field. *Term_no* denotes the position of the term within the document. For ease of implementation we again take it to be the beginning line number of the term. As in the inverted tag index, *level* stores the nesting depth of the term within the document.

4.2. Query evaluation via the extended inverted indexing technique

This is an indexing technique for generic XML documents. So, like other generic indexing techniques the peculiarities of product catalogs, discussed in Sections 2 and 5, and the most frequently asked query types over product catalogs, discussed in Section 3, are not considered. Therefore all the four types of queries are processed in the same way. Another drawback is that the number of join operations performed to evaluate a query is proportional to

Table 1
Inverted Tag Index for a portion of the sample product catalog in Appendix A

Element	start_offset	end_offset	Level
Catalog	1	207026	0
Category	2	106	1
Name	2	2	2
Product	3	25	4

Table 2
Inverted Term Index for a portion of the sample product catalog in Appendix A

Term	term_no	Level
Books	2	3
1	3	6
The Brothers Karamazov	4	6
Fyodor Dostoyevsky	6	6

the path length of the query. This leads to a serious problem for large XML documents. As an example let us evaluate query Q1 (/catalog/category[@name = 'books']/product[name = 'The Brothers Karamazov']) of Type 1. As seen below six join operations are performed:

- 'catalog' and 'category' (retrieve category elements who have catalog element as parent)
  self-join on InvertedTagIndex
  join condition:
    catalog.start_offset ⩽ category.start_offset and
    catalog.end_offset ⩾ category.end_offset and
    catalog.level = category.level - 1
- 'category' and 'name' (retrieve name attributes who have category element as parent)
  self-join on InvertedTagIndex
  join condition:
    category.start_offset ⩽ name.start_offset and
    category.end_offset ⩾ name.end_offset and
    category.level = name.level - 1
- 'name' and 'books' (retrieve name elements with value 'books')
  join InvertedTagIndex with InvertedTermIndex
  join condition:
    name.start_offset ⩽ books.term_no and
    name.end_offset ⩾ books.term_no and
    name.level = books.level - 1
- 'category' and 'product' (retrieve product elements who have category element as parent)
  self-join on InvertedTagIndex
  join condition:
    category.start_offset ⩽ product.start_offset and
    category.end_offset ⩾ product.end_offset and
    category.level = product.level - 1
- 'product' and 'name' (retrieve name attributes who have product element as parent)
  self-join on InvertedTagIndex
  join condition:
    product.start_offset ⩽ name.start_offset and
    product.end_offset ⩾ name.end_offset and
    product.level = name.level - 1
- 'name' and 'The Brothers Karamazov' (retrieve name elements with value 'The Brothers Karamazov')
  join InvertedTagIndex with InvertedTermIndex
  join condition:
    name.start_offset ⩽ The Brothers Karamazov.term_no and
    name.end_offset ⩾ The Brothers Karamazov.term_no and
    name.level = The Brothers Karamazov.level - 1

5. Indexing technique for XML-based product catalogs

We designed the proposed efficient indexing structure for XML-based electronic product catalogs by considering the most frequently asked queries over product catalogs, discussed in Section 3; and the typical data stored in a typical product catalog, as discussed in the following.

Since, there is no approved standard that agrees on the common tags with common semantics for XML-based electronic product catalogs, each vendor names product attributes with different semantics. This poses a big challenge in designing an efficient storage structure for XML-based product catalogs. However, by examining different product catalogs and different product types, we have decided that the most flexible and presentable way to organize product catalogs is to group products into categories. All the products have some common properties such as category name, product name, product description, keywords, product price, on sale date, shipping information, image or clip of the product. Thus, we have decided to index this information with the exception of the image and clip information because they are not searchable. On the other hand, each product may have some properties specific to itself such as author or publication information for a book and hardware information for a computer. In our indexing scheme we also index this type of information specific to a product.

5.1. The structure of the proposed indexing technique

We use four types of indexes. The indexes are stored as tables in a relational DBMS for implementation simplicity. In the following subsections we discuss the four index types.

5.1.1. Category index

In electronic commerce applications, category searches are very common. Customers very often query to view all the products associated with a specific category. For instance a customer may want to view all the books or all the CD's present in the product catalog (queries of Type 3). Thus, we decided to have a separate category index. In Table 3, category index for a portion of the sample product catalog in Appendix A is given.

To create the category index, the XML-based product catalog is parsed and each path of a category is stored in the form of XPath [23] expression in the *category_path* field of the category index. In the *category_start_offset* and *category_end_offset* fields, the start offset and end offset of the category element in the XML file are stored.

Table 3
Category Index for a portion of the sample product catalog in Appendix A

category_path	category_start_offset	category_end_offset
/catalog/category[@name = 'books']	2	106
/catalog/category[@name = 'CDs']	111	163
/catalog/category[@name = 'widgets']	169	7073

Table 4
Path Index for a portion of the sample product catalog in Appendix A

Path	path_id
/catalog/category[@name = 'books']/product/name	1
/catalog/category[@name = 'books']/product/author	2
/catalog/category[@name = 'books']/product/publication	3
/catalog/category[@name = 'books']/product/publisher/name	4

5.1.2. Path index

In Table 4, Path Index for a portion of the sample product catalog in Appendix A is given.

To create this index the XML-based catalog is traversed and the paths that have attribute or element values are inserted in XPath format [23] as string values into the *path* field of the path index. A unique ID is assigned to each different path and stored in the *path_id* field.

5.1.3. Value index

The value index stores the value of each element or attribute in the product catalog as a string in the *value* field of the index; the product ID of the product which this element or attribute belongs in the *product_id* field; and the path ID of the XPath path leading to this element or attribute in the *path_id* field. In Table 5, we give a portion of the value index for the product catalog a part of which is given in Appendix A.

To create the value index the XML-based product catalog is parsed and the path ID of the path leading to the specific element is retrieved from the path index. This path ID is stored in the *path_id* field of the value index. The value of the element is stored in the *value* field of the value index and the ID of the product which this element or attribute belongs is stored in the *product_id* field of the value index. For instance, the path leading to the element value "The Brothers Karamazov" is: '/catalog/category[@name = 'books']/product/name'. The *path_id* of this path is obtained from the path index and is 1. The product ID of the product with name "The Brothers Karamazov" is 1. This forms the first entry in the value index.

5.1.4. Product index

In the product index, the unique product ID of each product is stored in the *product_id* field of the index. The start and end offsets of each product element are stored in the *start_offset* and *end_offset* fields of the product index. To construct the product index the XML-based product catalog is parsed using the SAX (Simple API For XML) standard and the start and end offsets of each product element together with the product ID are inserted to the product index. In Table 6, product index for a portion of the sample product catalog in Appendix A is given.

Table 5
Value Index for a portion of the sample product catalog in Appendix A

path_id	Value	product_id
1	The Brothers Karamazov	1
2	Fyodor Dostoyevsky	1
3	1	1
4	Penguin classics	1
5	1993	1

Table 6
Product Index for a portion of the sample product catalog in Appendix A

product_id	start_offset	end_offset
1	3	25
2	26	48
3	49	74
4	76	104
5	112	129

5.2. Query evaluation via the proposed indexing technique

The implementation of the storage and indexing structure of the XML-based electronic product catalog is done using JAVA. The queries are processed using XPath [23]. The original XML document is parsed using the SAX model (which is used only once while creating the indices). While processing a query, first the query is parsed and the path ID of the path is found from the path index. The product ID and element value are returned according to the path ID from the value index. The portion of the XML-based product catalog that stores the relevant product or products is returned using the start offset and end offset obtained from the product index.

In electronic commerce applications, it is required to return the result of a query in XML format so that it can be presented to the customer with an appropriate style sheet. Therefore, in the system we developed, when there is a lookup operation for specific products with specific properties, the matched product elements are returned in XML format (queries of Type 1). This is the reason why the start and end offsets of product elements are stored in the product index. If specific values need to be returned, such as the names of all books (queries of Type 2 or Type 4), they are returned directly from the index structure, without accessing the original XML file. Since the queries such as listing all the products in a specific category are very common and frequent (queries of Type 3), the separate category index is useful. When the list of all the products of a specific category is queried, the portion of the XML document specified by the category_start_offset and category_end_offset is returned. Storing start and end offsets of products and categories, reduces the portion of the XML-based product catalog file to be scanned.

Since the paths are stored as strings in XPath format, path matching is a simple string matching operation. Thus, extensive join operations such as in the extended inverted index are not performed. Furthermore, the path in the submitted XPath query need not start from the root and it need not consist of just direct parent–child relationships ('/' operator) but it can also contain ancestor–descendant relationships ('//' operator). In these cases partial string matching is performed.

5.2.1. Evaluating queries of Type 1

Suppose we want to evaluate query Q1 (/catalog/category[@name = 'books']/product[name = 'The Brothers Karamazov']) of Type 1 (type of queries for retrieving products with specific given properties). The following operations are performed:

- join PathIndex with ValueIndex to obtain the product_ids of the products with the specified properties
  join condition:
    PathIndex.path = "/catalog/category[@name = 'books']/product/name" and
    PathIndex.path_id = ValueIndex.path_id and
    ValueIndex.value = "The Brothers Karamazov"
- join ValueIndex with ProductIndex to obtain the start_offsets and end_offsets of the products with the product_ids obtained from the previous join operation
  join condition:
    ProductIndex.product_id = ValueIndex.product_id
- Return the portion of the XML-based product catalog between the start_offsets and end_offsets found in the previous step

Only two join operations are performed. Only the portion of the XML-based product catalog between the obtained start-offsets and end-offsets is scanned.

5.2.2. Evaluating queries of Type 2

Suppose we want to evaluate query Q5 (catalog/category[@name = 'books']/product/name), that retrieves the list of the names of the books in the product catalog, of Type 2 (type of queries for retrieving all the values of a specific given property of products of certain type). The following operations are performed:

- join PathIndex with ValueIndex to obtain the values of the given property (names of the books in the catalog)
  join condition:
    PathIndex.path = "/catalog/category[@name = 'books']/product/name" and
    PathIndex.path_id = ValueIndex.path_id
    Return ValueIndex.value

Only one join operation is performed. The XML-based product catalog file is not accessed at all. The result of the query is returned directly from the *path index* and the *value index*.

5.2.3. Evaluating queries of Type 3

Suppose we want to evaluate query Q7 (/catalog/category[@name = 'CDs']), that lists all the products in the CD's category, of Type 3 (type of queries for retrieving products of a certain given category). The following operations are performed:

- Retrieve CategoryIndex.category_start_offset and CategoryIndex.category_end_offset of the category where
    CategoryIndex.category_path = "/catalog/category[@name = 'CDs']"
- Return the portion of the XML-based catalog file between CategoryIndex.category_start_offset and CategoryIndex.category_end_offset obtained in the previous step.

Queries of this type are very common in product catalogs and they are processed very efficiently by our indexing structure. Only the *category index* is used and no join operations are performed. Only the portion of the XML-based product catalog between the obtained category_start_offset and category_end_offset is scanned.

5.2.4. Evaluating queries of Type 4

Suppose we want to evaluate query Q8 (/catalog/category[@name = 'books']/product[name = 'The Brothers Karamazov']/price), that retrieves the price of the book with name "The Brothers Karamazov", of Type 4 (type of queries for retrieving certain properties of products with specific given properties). The following operations are performed:

- join PathIndex with ValueIndex to obtain the product_ids of the products with the specified properties
  join condition:
    PathIndex.path = "/catalog/category[@name = 'books']/product/name" and
    PathIndex.path_id = ValueIndex.path_id and
    ValueIndex.value = "The Brothers Karamazov"
    Return ValueIndex.product_id
- join PathIndex with ValueIndex to obtain the value of the asked property
  join condition:
    PathIndex.path = "/catalog/category[@name = 'books']/product/price" and
    PathIndex.path_id = ValueIndex.path_id and
    ValueIndex.product_id = product_id obtained in the previous step
    ValueIndex.value = "The Brothers Karamazov"
    Return ValueIndex.Value (price of the book)

Only two join operations are performed. The XML-based product catalog file is not accessed at all to obtain the result. The result is obtained directly from the *value index* and the *path index*.

6. Performance study

In electronic commerce applications, the traditional way of executing queries over XML-based product catalogs is by storing the document in its original format and using SAX (Simple API for XML) or DOM (Document Object Model) API to execute the queries. DOM reads an XML document and returns a representation of this document as a tree of nodes. Queries are executed by traversing the tree. The tree is memory resident during the query execution and constructed each time a query is executed. SAX, on the other hand is an event-driven API which accesses the XML document through a sequence of events. SAX reads the document and reports parsing events (such as the start and end of element tags). SAX reads the information much faster than DOM, since it does not need to allocate the tree structure. In addition SAX requires less memory [19].

In the method we propose, the XML document is parsed only once in order to create the indexes. The indexing engine runs also when updates occur. However, updates are very rare in product catalogs as discussed earlier and stated in [20]. The queries are executed over the indexes. The proposed system is compared to the traditional system as a baseline, where queries are executed over the XML document using the SAX API. SAX is chosen instead of DOM for the performance analysis and comparison because it is more efficient and suitable than DOM for large product catalogs that do not fit into main memory. As discussed earlier, the inverted index technique [16] extended for XML documents and used for XML-based product catalogs in [17] is also used for comparison. JBuilder 6.0 and JDK 1.4.0 are used for the implementation and test runs. SAX API of java.sun.com is used as the parser and for the test runs of the traditional method. All the experiments were performed on an Intel Pentium IV 1.5 GHz machine with 128 MB of memory and 40 GB of local disk storage, running under Windows 2000 Professional. To store and access the index structures for the extended inverted index and for the proposed indexing technique MS Access relational DBMS is used. In the remainder of this section the product catalog documents in XML format and the queries used in our experiments are described. Finally the experiment results are displayed and evaluated.

6.1. Document set

In the performance study four XML-based product catalog documents are used. In Table 7, the important features of the four documents are listed.

6.2. Queries used in the performance study

In the Section 3, we have classified the queries executed over product catalogs into four groups. In the performance study query execution times of each type of query are measured for the proposed indexing technique, extended inverted indexing technique [16,17] and for the traditional technique. In Table 8, the main query types executed over product catalogs and the queries used for the experiments are summarized.

Table 7
The datasets used for the experiments

	Catalog1	Catalog2	Catalog3	Catalog4
Size of the document (MB)	11	30.9	60.7	120
Number of product elements	15,000	42,000	82,500	163,500
Number of category elements	12	30	57	111

Table 8
Query classification

Type 1 queries	Q1	/catalog/category[@name = 'books']/product[name = 'The Brothers Karamazov']
	Q2	/catalog//product[price[@currency = '$'] < 15]
	Q3	/catalog/category[@name='books']/product[name= 'The Brothers Karamazov' and author= 'Fyodor Dostoyevsky']
Type 2 queries	Q4	//category[@name = 'books']//author
	Q5	/catalog/category[@name = 'books']/product/name
Type 3 queries	Q6	/catalog/category[@name = 'books']
	Q7	/catalog/category[@name = 'CDs']
Type 4 queries	Q8	/catalog/category[@name = 'books']/product[name = 'The Brothers Karamazov']/price
	Q9	/catalog/category[@name='widgets']/product[name= 'umbrella']/description

6.3. Experimental results

Test runs have been done over the documents described in Section 6.1 for the queries discussed in Section 6.2. The query processing time is used as the performance metric. Table 9 (where P stands for the Proposed Indexing Tech-

Table 9a
Summary of performance results (in ms) for datasets catalog1 and catalog2

Datasets	Catalog1			Catalog2		
Query	P	I	T	P	I	T
Q1	361	1091	1873	381	2995	3705
Q2	1191	1282	2584	1953	2634	4276
Q3	401	1652	1853	611	3575	3735
Q4	421	941	1773	751	2454	3455
Q5	440	1031	1742	741	2654	3845
Q6	921	1933	2914	1281	3044	4566
Q7	591	1352	2223	742	3004	4036
Q8	100	1472	1622	110	3555	3745
Q9	90	1433	1643	101	3435	3876

Table 9b
Summary of performance results (in ms) for datasets catalog3 and catalog4

Datasets	Catalog3			Catalog4		
Query	P	I	T	P	I	T
Q1	1252	4887	6750	2273	9903	12,809
Q2	5750	5458	7360	7861	9352	13,719
Q3	861	6620	6789	1141	11,004	12,568
Q4	3525	4266	6339	6088	8141	14,381
Q5	3505	4647	6880	6019	8721	13,509
Q6	1322	5357	7231	1692	10,111	12,057
Q7	801	5147	6639	881	9758	11,927
Q8	301	6369	6430	401	11,547	12,338
Q9	290	7000	6920	380	11,797	12,849

nique, I stands for the Extended Inverted Index Technique and T stands for the Traditional Technique) summarizes the results obtained from the experiments.

We present the graphical summary of the performance results for documents catalog1, catalog2, catalog3 and catalog4 in Figs. 1–4, respectively.

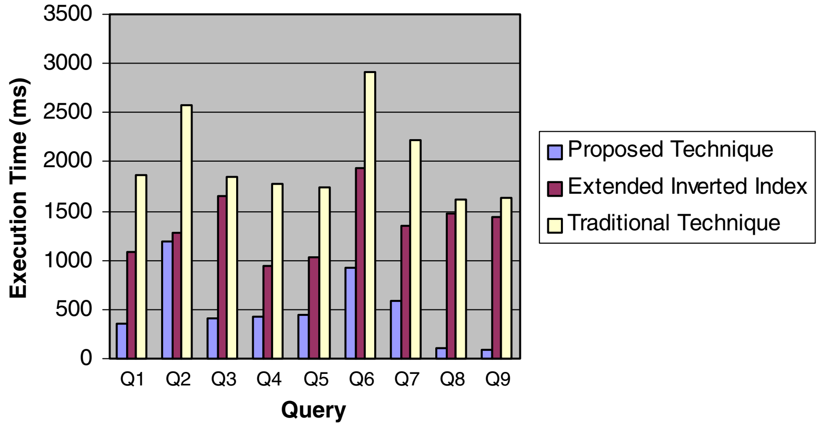

Fig. 1. Experiment results over catalog1.

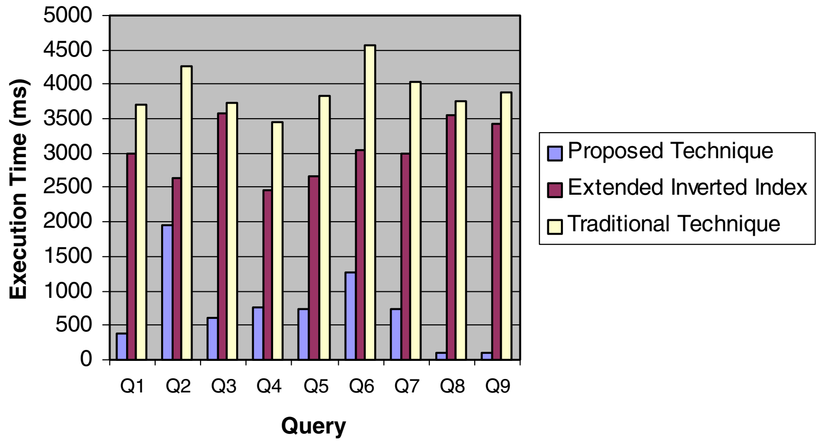

Fig. 2. Experiment results over catalog2.

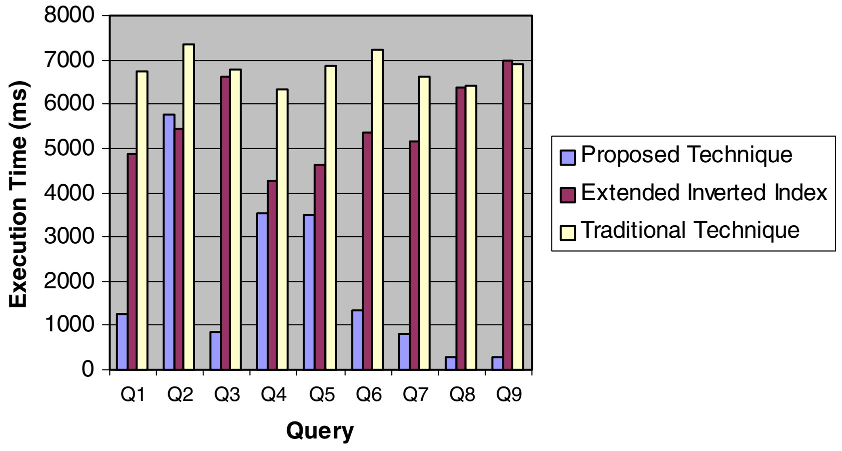

Fig. 3. Experiment results over catalog3.

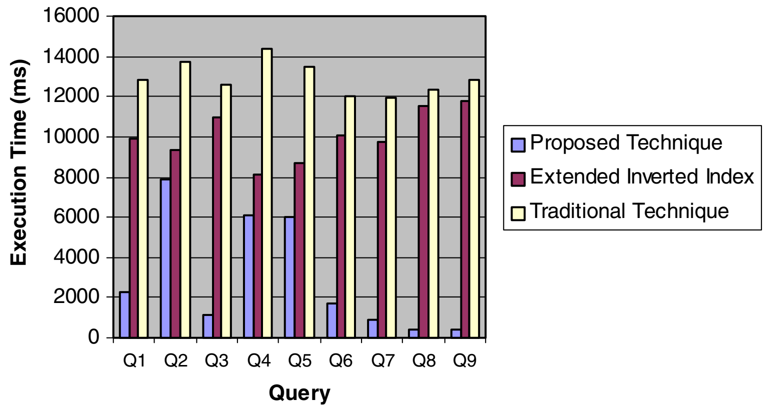

Fig. 4. Experiment results over catalog4.

It is revealed that the proposed system is more efficient than the traditional system and the extended inverted indexing technique for all of the main types of queries executed over product catalogs. Especially for the queries of Type 3 and Type 4, the proposed technique outperforms considerably the extended inverted indexing technique and the traditional system. The reason is that there is a separate category index in the proposed system for Type 3 queries. For Type 4 queries in the proposed system access to the original XML file is not needed, i.e., results are obtained directly from the index structure. This is valid also for the extended inverted index technique. However, in the extended inverted index technique, the overall path structure is not stored. Therefore, the number of join operations is proportional to the path length of the query evaluated. In the proposed technique, the paths are stored directly as XPath expressions and thus path matching is a simple string matching operation. For instance for query Q1, six join operations are performed with the extended inverted indexing technique, while only two join operations are performed with the proposed technique. With the proposed indexing method to execute queries of Type 1 and Type 4, two join operations are performed; to execute queries of Type 2, only one join operation is performed; and no join operations are performed to execute queries of type 3. Thus the extended inverted index technique has considerably worse performance for lengthy queries. Also, extended inverted index technique does not consider the specific properties and needs of product catalogs and consequently indexes all the elements and attributes and their values. This increases the index size considerably and also degrades its performance. In the traditional system, parsing the document and navigating thorough the whole document each time a query is asked decreases the performance drastically.

In this performance study, we assumed that product catalogs are well-organized and have a hierarchical structure of categories and sub-categories. However, if the users want to express their products without product categories or product category as an attribute of a product, they can use the indexing scheme again, but without the category index which speeds Type 3 queries. The indexing scheme can be used again to speed Type 1, 2, and 4 queries. We can conclude that our indexing scheme increases especially the performance of the four types of queries, but can be used for other types of queries and for catalogs where products are not organized into categories as well.

7. Conclusion

Efficient processing of queries posed over product catalogs is important for customer satisfaction. Storing product catalogs in XML provides some advantages such as interoperability and flexibility. Due to XML's increasing popularity, we expect that product catalogs will be commonly stored in XML in the near future. One of the basic approaches in storing XML documents is to keep XML documents in their original form. Keeping XML documents in their original form eliminates the burden of transforming XML into other formats and back again to XML format for data exchange. Also processing of natively stored XML documents is rapidly gaining momentum in the literature and practice.

In this paper, we present a novel technique for indexing product catalogs stored in native XML format. The technique we propose considers the peculiarities of product catalogs in general and the commonly asked queries on product catalogs. We compare the performance of our proposed technique with the traditional SAX-based retrieval as a baseline and the extended inverted indexing technique proposed for product catalogs in [17]. As far as we know the extended inverted indexing technique is the only one proposed for product catalogs in the literature. Our technique outperformed the baseline method and the extended inverted indexing technique for all categories of queries posed over natively stored XML product catalogs of various sizes.

Appendix A

A sample portion of a product catalog in XML:

```
<catalog>
   <category name=”books”>
        <product id=”1”>
             <name>The Brothers Karamazov</name>
             <description>A World Classic</description>
             <author>Fyodor Dostoyevski</author>
             <publication>1</publicationõ>
             <publisher>
                  <name>Penguin Classics</name>
                  <date>
                         <year>1993</year>
                         <month>02</month>
                   </date>
                  <city>London</city>
             </publisher>
             <price currency=”$”>12</price>
             <quantity_in_stock>100</quantity_in_stock>
             <image format=”gif” width=”234” height=”400”           src=”images/karamazov.gif”/>
             <on_sale_date>
                       <year>1999</year>
                       <month>03</mo nth>
                       <da y>04</day>
               </on_sale_date>
             <shipping_info>cargo</shipping_info>
        </product>
        <product id=”2”>
               .
               .
               .
        </product>
        <product id=”3”>
               .
               .
               .
        </product>
        <product id=”4”>
               .
               .
               .
        </product>
</category>
<category name=”CDs”>
       <product id=”5”>
               .
               .
               .
        </product>
               .
               .
               .
</category>
<category name=” widgets”>
               .
               .
               .
</category>
</catalog>
```

References

[1] T. Bray, J. Paoli, C.M. Sperberg-McQueen, Extensible Markup Language (XML) 1.0. Available from: <http://www.w3.org/TR/REC-xml>.

[2] F. Tian, D. DeWitt, J. Chen, C. Zhang, The design and performance evaluation of alternative XML storage strategies, ACM SIGMOD Record 31 (1) (2002).

[3] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, XML-QL: a query language for XML, in: Proceedings of the International WWW Conference, 1999.

[4] D. Chamberlin, D. Florescu, J. Robie, J. Sim'eon, M. Stefanescu, XQuery: A Query Language for XML, W3C, 2000. Available from: <http://www.w3.org/TR/xmlquery>.

[5] J. Robie, J. Lapp, D. Schach, XML query language (XQL), in: Proceedings of the Query Languages Workshop, Cambridge, MA, December 1998. Available from: <http://www.w3.org/TandS/QL/QL98/pp/xql.html>.

[6] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J.L. Wiener, The Lorel query language for semistructured data, Journal on Digital Libraries, 1997. Available from: <ftp://db.stanford.edu/pub/papers/lorel96.ps>.

[7] A. Deutsch, M.F. Fernandez, D. Suciu, Storing semi-structured data with STORED, SIGMOD Conference (1999) 431–442.

[8] D. Florescu, D. Kossman, A performance evaluation of alternative mapping schemes for storing XML data in relational database, Rapport de Recherche No. 3680 INRIA, Rocquencourt, France, May 1999.

[9] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, J.F. Naughton, Relational databases for querying XML documents: limitations and opportunities, VLDB (1999) 214–302.

[10] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, Lore: A database management system for semi structured data, SIGMOD Record 26 (3) (1997) 54–66.

[11] F. Bancihon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, F. Velez. The design and implementation of O2, an object-oriented database system, in: K. Dittrich (Ed.), Proceedings of the Second International Workshop on Object-oriented Database, 1988.

[12] M. Carey, D. DeWitt, J. Naughton, M. Solomon, et al., Shoring up persistent applications, in: Proceedings of the 1994 ACM SIGMOD Conference.

[13] R. Goldman, J. Widom, DataGuides: enabling query formulation and optimization in semistructured databases, in: Proceedings of the International Conference on Very Large Databases, 1997, pp. 436–445.

[14] T. Milo, D. Suciu, Index structures for path expressions, in: Proceedings of the International Conference on Database Theory, 1999, pp. 277–295.

[15] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison-Wesley, Reading, MA, 1999.

[16] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On supporting containment queries in relational database management systems, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 2001.

[17] Ee-Peng Lim, Wee-Keong Ng, An overview of the agent-based electronic commerce system (ABECOS) project, Bulletin of The IEEE Computer Society Technical Committee on Data Engineering 23 (1) (2000) 49–54.

[18] A2i, Inc. Product catalogs-why a relational DBMS and SQL are not enough, Technical White Paper, version 1.03, May 9, 2001. Available from: <http://www.test.bitpipe.com/rlist/org/1029567262_495.html>.

[19] J. Ueyama and E.R.M. Madeira, Using XML for electronic catalogs, in: Workshop on Information Integration on the Web 2001: 43–50. Available from: <http://www.cos.ufrj.br/wiiw/papers/06-Jo_Ueyama(31).pdf>.

[20] RosettaNet Consortium, Official site. http://www.rosettanet.org/.

[21] CommerceNet, The eCo framework standardization project. Available from: <http://www.commerce.net/projects/currentprojects/eco/>.

[22] W.D. Smith, TPC-W: Benchmarking an E-Commerce Solution, Revision 1.2, Intel Corporation. Available from: <http://www.tpc.org/tpcw/TPC-W_Wh.pdf>.

[23] W3C Recommendation, XML Path Language (Xpath) 1.0, 1999. Available from: <http://www.w3.org/TR/xpath>.