Knowledge and Information Systems (2001) 3: 252–261

Knowledge and Information Systems

Short Paper

XML Indexing and Retrieval with a Hybrid Storage Model

Dongwook Shin

National Library of Medicine, Bethesda, Maryland, USA

Abstract. XML DTD (Document Type Declaration) puts two distinctive entities (attribute and element content) together into one framework for representing different document features. The notion of attribute in the XML DTD is similar to the field representation in the database, whereas the element content corresponds to the full text. In this paper, we view these two entities as different, each of which requires a different model for storage and retrieval. Attributes are stored in a database system, whereas the element contents and their indices are saved in files. We present a technique that puts together those two in an efficient way and builds an XML retrieval system on top of that. Such a system can achieve a reasonable trade-off between performance and cost in indexing and retrieval.

Keywords: Database system; Hybrid storage model; Information retrieval; XML

1. Introduction

The growing volume of XML contents (W3C, 1998) in the world has drawn attention to the document management tools that can manage them effectively. One core component in these management systems is the search engine that lets people find relevant content. XML search engines can be similar to World Wide Web search engines, most of which simply find the whole HTML documents. But they are also capable of providing structured search capabilities, allowing people to retrieve the relevant document elements no matter where they reside in the document structure.

In the design and development of XML retrieval systems, it is very important to characterize each entity in XML and embody it in an appropriate storage model. In this paper, we hypothesize that XML DTD (Document Type Declaration) puts two distinctive entities (attribute and element content) together in

Received 27 January 2000
Revised 29 July 2000
Accepted 3 November 2000

one framework. The notion of attribute in the XML DTD is similar to the field representation in the database, whereas the element content corresponds to the full text. In general, attributes represent exact concepts, which requires that a database system should be accurate in retrieving matched attribute. On the other hand, the full text employs natural language and the concepts expressed are often vague. Another difference is the scale (size of information) of the two. The number of words in the full text is usually far larger than the number of attributes appearing in a document. If we put the whole word list of a large XML collection into a database, which can reach billions of word occurrences, it may exceed the volume that a database system can effectively handle. In contrast, if we put the attributes into files as well as indices for the full text, we have to implement necessary database operations including comparison and join operations, which may require substantial extra work.

In this paper, we view these two entities as different from each other and use different storage models in storing and retrieving them. That is to say, attributes are stored in a database system, whereas the element contents and indexes are saved as flat files. With this premise, we develop XRS-II (XML Retrieval System), which is able to do a variety of structural and attribute searches in any document structure. XRS-II extends XRS-I (Shin, 1999), which was only able to handle the full text. It employs a Java-based relational database system (Instant DB, 1999) for storing and retrieving attributes, while putting original documents and inverted indices into flat files as in XRS-I.

In indexing and retrieving the full text, XRS-II employs the BUS (Bottom-Up Scheme) technique (Shin et al, 1998). BUS does indexing only at the element including full text, whereas the index information of the intermediate elements is computed at retrieval.

Secondly, XRS-II uses an XSLT processor (W3C, 1999a) that renders the XML output into HTML. It makes it possible for a user without an XML-aware browser to view the search results.

2. System Architecture

XRS-II is composed of four components. The first one is a GUI written in Java applet, whereas the second is the Query Mediator servlet (Hunter, 1998), which mediates messages between users and the back-end search engine. The third is the XSL (W3C, 1999a) processor, which transforms the XML content into HTML. The fourth is the back-end search engine, which uses a full-text search and a relational database. Figure 1 shows the whole architecture and the relationship among its components.

First, when a user accesses XRS-II via the Web, a user interface is spawned from the Web page that establishes a communication session between the user and the system. The Query Mediator servlet running on the server keeps track of the active sessions. It sends user queries to the back-end search engine and passes the results back to the user interface or the XSL processor, depending on the type of results. When a user issues a query and the search engine produces a set of element surrogates (element head information with similarity value) relevant to the queries, the set is directly sent to the GUI. If a user wants to get an element content among the set and the search engine gives back the XML element, it goes through the XSL processor, which transforms the XML into HTML according to the style program written in XSL.

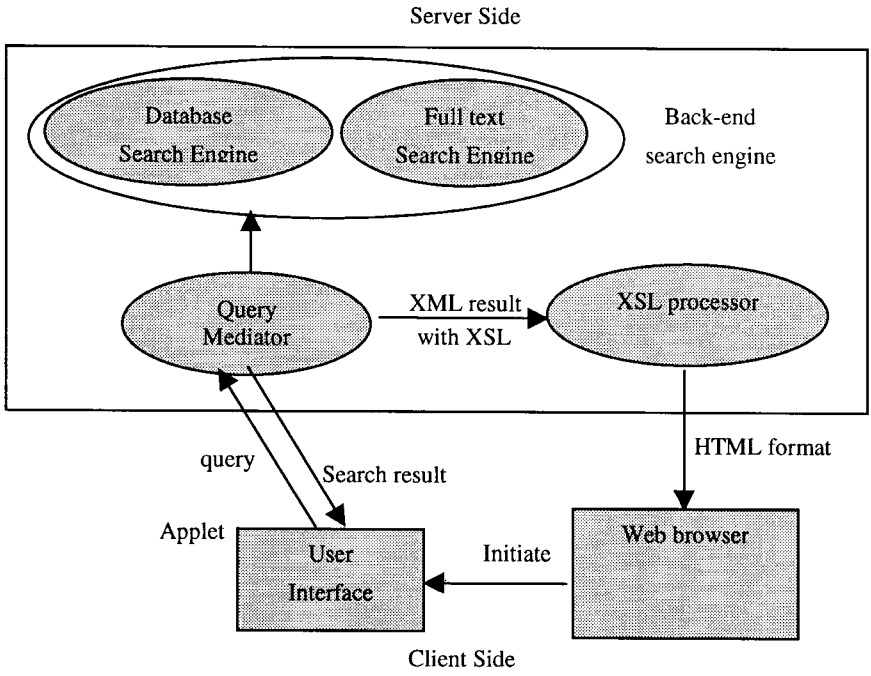

Fig. 1. The XRS-II system architecture.

The back-end search engine processes user queries and retrieves the relevant elements. It is composed of two components: database search and full-text search engine. The database search engine does attribute matching which retrieves only the exact matched elements against the given condition. In contrast, the full-text search engine searches contents in full text (text appearing in PCDATA or CDATA section) and retrieves the relevant elements no matter where the element is in the document structure. These two engines work in parallel and create their own results, which are eventually merged together to produce the final results.

3. Text Indexing and Searching

In order to index text and search queries efficiently, we apply BUS (Shin et al, 1998) to the full-text in XML. The main idea is to do indexing only at the elements including full text (the elements having text nodes as direct children). The indices for internal nodes without full text are not created at index time, but are reproduced dynamically at retrieval. That is to say, term frequencies are collected only at the elements having full text and saved in the inverted file (Harman et al, 1992). If a user wants to retrieve nodes at an intermediate level, all the term frequencies of text nodes nested in the intermediate node are accumulated together, which result in the whole term frequency in the intermediate node.

In order to facilitate the accumulation of term frequencies in the corresponding internal nodes, Shin et al (1998) introduced the notion of GID (General element IDentifier), extending that of UID (Unique element IDentifier) proposed by Lee et al (1996). The UID scheme is to represent a document as a k-ary complete

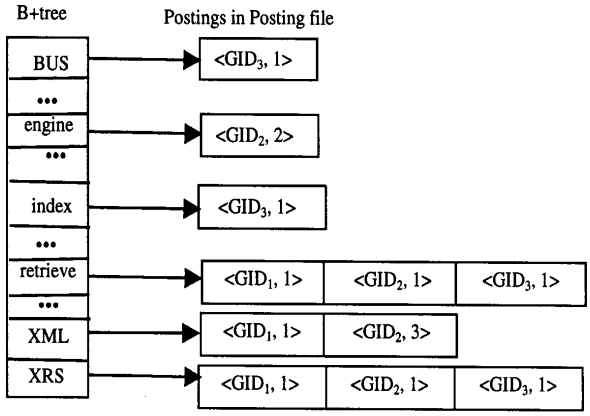

Fig. 2. A posting structure in XRS-II.

tree, where k is the largest number of siblings in the structure. The result of the mapping is called the '*document tree*'. Each element is assigned a UID according to the order of the level-order tree traversal. In this tree, with the knowledge of a child's UID, one can compute the parent UID easily using the following expression:

$$\text{parent\_UID}(\text{child\_UID}) = \frac{(\text{child\_UID} - 2)}{k} + 1 \tag{1}$$

where k is the largest number of siblings in the tree.

Extending the UID scheme, BUS assigns a unique GID to each node. A GID consists of (1) document number (DID), (2) the UID of the element in the document tree, (3) the level (LEV) of the element in the document tree, and (4) the element type number (EID) in the structure. The DID indicates which document the element belongs to and the UID tells the element location in the document tree. LEV and EID facilitate the reproduction of term frequencies in the appropriate level that a user wants. That is, with LEV, we can compute the difference of the *target level* (the level of the elements that the user wants to retrieve) and the *index level* (the levels of the elements having full text and thus indexing is performed). The elements at the index level can then be merged to the elements at target level. EID is a unique number assigned to each element type name (element name appearing in DTD). It is utilized to filter out postings whose EIDs do not match those in user queries. The query language is explained in Section 5.

In text indexing, each word is associated with the corresponding GID. Going through the inversion process, the index is eventually made as the B+ tree and posting file (Harman et al, 1992). Figure 2 illustrates the posting structure from the text in Example 4.1.

In performing the query evaluation procedure (QEP), BUS calls for as many memory cells (called accumulator) as the number of elements in the target level, each of which is assigned to an element in the level. It accumulates all the term frequencies appearing in the nested elements. Term frequencies in the leaf

elements can be accumulated to the corresponding elements at target level, since we can calculate the UID of target element from the LEV and UID by repeatedly applying expression (1).

The QEP accumulates term frequencies from the postings and filters out the unnecessary postings whose EIDs are not descendants of the target node. For instance, in Example 1, if we want to retrieve PARs having 'XRS', the postings whose EIDs are 'TITLE' should not participate in the frequency accumulation. This is because any occurrence of 'XRS' in TITLE should not be accumulated in counting the number of occurrences in PAR.

4. Attribute Indexing and Searching

As mentioned previously, we put the attributes into the database tables in such a way that all the attribute lists pertaining to the same element type are saved in the same table. In another words, a table is allocated to each element type if it has an attribute list.

Example 4.1. (A sample XML instance). As the document has two different attribute lists, one for 'AUTHOR' and the other for 'PAR', XRS-II saves the attributes into two different tables

```
<DOC>
  <DATE>1999-10-15</DATE>
  <TITLE>XRS: XML retrieval system</TITLE>
  <AUTHOR LAST='Shin' FIRST='Dongwook'>
  </AUTHOR>
  <ABSTRACT>
    <PAR id='1'>
        XRS is an XML search engine that is able to retrieve any
        elements a user wants very
        effectively. Unlike other XML search engines that get
        back whole XML documents to
        you, you can impose conditions on any elements with
        weights and get back back relevant...
    </PAR>
 <PAR id='2'>
        XRS uses a couple of new techniques that have been
        recently developed. One of those
        is the BUS (Bottom Up Scheme) technique developed for
        indexing and retrieving structured documents efficiently.
  </PAR>
</ABSTRACT>
...
</DOC>
```

Attribute Table for /DOC/AUTHOR

DID	UID	LEV	Last	First
…	…	…		…
1	4	2	Shin	Dongwook
…	…	…	…	…

Attribute Table for /DOC/ABSTRACT/PAR

DID	UID	LEV	Id
…	…	…	…
1	22	3	1
1	23	3	2
…	…	…	…

Fig. 3. Attribute index table.

/DOC/AUTHOR

/DOC/ABSTRACT/PAR

EID	DID	UID	LEV	ATT NAME	ATT VALUE
…	…	…	…	…	…
4	1	4	2	First	Dongwook
4	1	4	2	Last	Shin
6	1	22	3	Id	1
6	1	23	3	Id	2

Fig. 4. Another design for attribute index table.

Figure 3 shows two database tables, each of which corresponds to an attribute list in Example 1. If there is another 'AUTHOR' or 'PAR' in the XML document, it occupies a row in the corresponding table. Note that the number of tables is as many as the number of element types (or the number of EIDs) having attributes.

Another possible way to put attributes into tables is to create just one table and put all the attributes into the table. This raises a problem that each attribute list in an XML document has its own attribute name and value pairs. One solution may be to put each attribute name and value pair into each row in a table as in Fig. 4. However, this results in huge space overhead since there is significant redundancy in representing <EID, DID, UID, LEV> in each row if an attribute has more than one attribute name and value pair.

The QEP (Query Evaluation Procedure) for attributes is quite similar to that for full text. Attribute queries are first transformed into the SQL queries and run against the corresponding attribute tables. For instance, in the table for '/DOC/AUTHOR' in Fig. 3, the record '<1,4,2, Shin, Dongwook>' means that an attribute whose last name is 'Shin' and first name is 'Dongwook' appears in the element with the UID 4 in the first XML document. The level of the element is 2 since 'AUTHOR' is just below the root element 'DOC' whose level is 1.

If a user wants to retrieve an <ABSTRACT> that has a <PAR> whose *id* is '2', we should first access the attribute table for 'PAR' and retrieve the records whose 'id's are '2'. In Fig. 3, only a record whose DID, UID and LEV is 1, 23, and 3 respectively is retrieved, since its id is equal to '2'. Secondly, the QEP executes the parent function (expression 1) once and figures out the target UID is 5 (assume that k is 5). This is because the element having the attribute locates at level 3 from the 'LEV' column, but the user wants to get the elements at level 2.

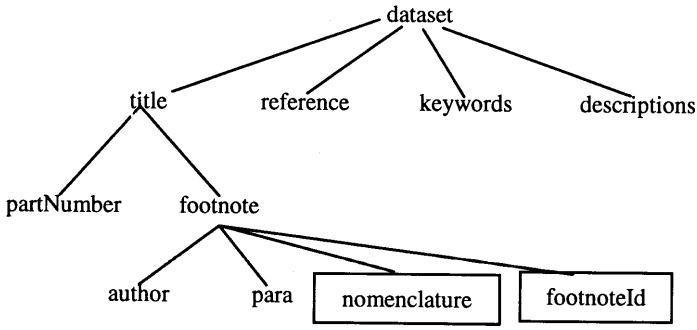

Fig. 5. A part of the ASTRONOMY DTD structure.

An advantage of using a database in processing attributes is that it can handle a set of comparison operators (greater than, less than, equal, not equal and so on) and join operators efficiently. For instance, if we want to retrieve ids greater than '10', we can do so with a simple SQL command (Lans, 2000) and retrieve the matched records quickly. We do not need to write extra codes to do that. Moreover, it is not easy to beat the relational database system in terms of performance.

If a user query is mixed with text and attributes, the steps addressed in both Section 3 and 4 are performed together and the results are merged. In merging, all the elements retrieved from the database are parts of the search results and those from the full-text play the role of carrying the similarity values to the elements extracted from the database. For instance, consider a query 'retrieve <PAR> whose 'id' is 1 and that contains 'XML' in Example 1. Note that the id of the first <PAR> is 1 among two <PAR>s. Hence only the first <PAR> is retrieved with the similarity value '3' ('3' comes from the frequency of 'XML' in the first <PAR>). The value '3' is used to rank the retrieved elements.

5. Experiment

As the experiment, we used three collections, two of which do not have attributes, and one of which (the astronomy data collection) does (NASA, 1999). Astronomy XML data is published from NASA ADC Center (NASA, 1999), and has a fairly complicated document structure with substantial attributes. Figure 5 shows a part of the astronomy DTD. Here, names contained in boxes represent attributes, whereas the others mean element names.

The astronomy XML data amounts to 33 Mbytes, with each document saved into a separate file. We first measured the statistics for the indexing. The number of postings is 696,065, whereas the number of attributes is 52,569. We use the Porter algorithm (Frakes, 1992) as the stemmer and 400 stop lists, which reduces the number of postings significantly. Note that the number of postings is 13 times larger than that of attributes even though we tried to reduce the number of postings by stemming and deleting stop words.

We wrote the programs in C and Java and tested the performance on a Sun Ultra-2 workstation with 256 Mbytes main memory. Table 1 summarizes the indexing performance for three data collections. The first two do not have

Table 1. A summary of index overhead.

Collection	Data size (Mbytes)	Index overhead			Time (h/min)
		Text index size (Mbyte)s	Attribute Index size	Index overhead (%)	
SHAKESPEARE	7	2.8	0	40.0	</02
CLINICAL	3	1.36	0	45.33	</01
ASTRONOMY	33	5.3	23	85.8	2/00
(without attribute)		5.3	0	15.8	/43

attributes and the indexing is performed fairly efficiently. The index overhead amounts to 40–45% of the original sizes with fast indexing time. On the other hand, the third collection has a substantial number of attributes. The index overhead for the full text is around 5.3 Mbytes, whereas that of the database amounts to 23 Mbytes. The reason why the index for the database occupies far more space than that of the full text is that a record takes hundreds of bytes depending on how many attributes it has, whereas in the inverted index a posting is saved as a compressed form. Even though the index space for attributes is relatively higher than that in full text, the whole space overhead is well below the original data size. But if we save the full text into a database, the space overhead can be 500–1000% of the source size since the number of postings is 13 times larger than that of attributes. Moreover it is very limited in applying the compression technique because the compression puts the postings into binary form and brings a substantial overhead in indexing and retrieval compared to the key-based retrieval in relational database systems.

Similarly, index time takes longer as the collection has many attributes. For instance, the astronomy collection takes around 2 hours to index whereas the same data without attributes takes 45 minutes. This is because every insertion of a record into a database goes through transaction processing, which brings a substantial overhead.

To analyze the querying performance, we implement a subset of XPath (W3C, 1999b) in such a way that it covers only the full-text and attribute searching part of the original XPath. XPath is a node selection language for XML and used as the selection part in XSL (W3C, 1999a). It composes queries as a sequence of path steps and predicates, which correspond to elements and conditions on elements, respectively. For instance, in an XPath query '*/dataset/descriptions*[*contains*(.,'*data*')]', '*/dataset/descriptions*' corresponds to two path steps and '[*contains*(.,'*data*')]' is a condition on an element '*descriptions*'. The query requires the retrieval of the '*descriptions*' elements of '*dataset*' that contain the word '*data*' in the full text. If '@' comes prior to a name, the name means an attribute instead of an element. For instance, in the query, '*/dataset*[@*subject*='*astronomy*']', '*subject*' is an attribute of the element '*dataset*'. In fact, the query calls for attribute searching since it wants to retrieve '*dataset*' elements whose '*subject*' attributes are equal to '*astronomy*'.

Table 2 shows the retrieval performance of the attribute and full-text searching when the queries are given as XPath forms.

Although the indexing overhead for attributes is rather high, attribute searching is faster than full-text searching. This is partly because the attributes are distributed over many tables depending on the element types (EID) that they

Table 2. A summary of the retrieval performance.

2.1 Attribute retrieval performance

Query	# of element retrieved	Time (s)
/dataset/fitsFile[@xlink:href="HD358.fit"]	1	0.5
/dataset/keywords/keyword[@xlink:href="Positional_data.html"]	106	0.4
/dataset[@subject="astronomy"]	1500	0.54
/dataset/altname[@type="ADC"]	1500	0.74
/dataset/tableHead/tableLinks/tableLink[@xlink:type="locator"]	5574	2.2

2.2 Full-text retrieval performance

Query	# of element retrieved	Time (s)
/dataset[contains(altname, "1005")]	1	0.6
/dataset/keywords/keyword[contains(., "Position")]	108	1.7
/dataset/descriptions/description/para[contains(.,"data")]	423	1.1
/dataset/descriptions/description/para[contains(., "star")]	574	1.3
/dataset[contains(.,"star")]	996	3.0

belong to. In fact, the attributes in the astronomy collection are dispersed into 25 tables, among which the biggest table has around 10,000 records. The second reason is that since each database table keeps an index for the key attribute, it is fairly fast to get attributes satisfying some conditions. On the other hand, in full-text retrieval, we have to get the postings from the posting file and check if each posting satisfies the condition one by one. Note that the retrieval takes longer as the number of elements retrieved is larger. One exception is the second full-text query, which takes longer than the third, even though its number of elements is smaller than that of the third. This is because the actual number of postings retrieved from the posting file is larger than that of the third, but most of the postings are removed since their EIDs do not match with the EID in the query (the EID of the element '*/dataset/keywords/keyword*'). In fact, most of postings for the word '*Position*' appear in the '*para*' element, not in the '*keyword*' element.

The use of a database in handling attributes brings another gain. It supports a couple of comparison operators and join operators efficiently. We do not need to write substantial codes corresponding to these operations.

6. Conclusion and Future Work

This paper proposes a hybrid storage model for implementing XML information retrieval. We view attributes and full text as different from each other and use different models for storing and retrieving them. Attributes are stored in a database system, whereas the element contents and indexes are saved into files. Consequently, we can achieve a reasonable trade-off between performance and cost in information retrieval.

As analyzed with a couple of collections, if word information is saved into database tables as well as attributes, it would take far more space than the original

data size. On the other hand, if all the attributes are saved into files similarly as postings, index space may be reduced. But we are responsible for making attribute search efficient with the support of a couple of comparison operators $(=, <, >)$ and the join operator. Compared to these, the hybrid model is an in-between approach, keeping the space, search time and development overhead in an allowable range.

Some future work remains. As mentioned before, the attribute indexing takes a rather long time since every insertion of each record into a table carries a transaction. However, as the indexing is usually performed off-line, it is not necessary to conduct a transaction for every insertion operation. We can make attribute indexing faster if we take the whole indexing as a transaction. In addition, with the analysis of attributes in the DTD prior to indexing, we can use a prepared statement instead of the normal insertion statement in SQL (Lans, 2000), which can also accelerate the indexing time.

Acknowledgements. The author is grateful to Dr Craig Locatis at the National Library of Medicine for valuable comments to the paper. He also appreciates Dr Alexa McCray for allowing him to continue the work and support the necessary things.

References

Frakes WB (1992) Stemming algorithms. In Frakes W, Baeza-Yates RA (eds). Information Retrieval: Data Structures and Algorithms. Prentice-Hall, Englewood Cliffs, NJ, pp 131–160

Harman D, Fox E, Baeza-Yates RA, Lee W (1992) Inverted files. In Frakes W, Baeza-Yates RA (eds). Information Retrieval: Data Structures and Algorithms. Prentice-Hall, Englewood Cliffs, NJ, pp 23–43

Hunter J (1998) Java servlet programming. O'Reilly and Associates, CA, USA

Instant DB (1999) A Java database engine [http://www.instantdb.co.uk]

Lans R (2000) Introduction to SQL: mastering the relational database language (3rd edn). Addison-Wesley, Reading, MA

Lee YK, Yoo SJ, Yoon K (1996) Index structures for structured documents. In Proceedings of Digital Library '96. ACM, New York, pp 91–99

NASA (1999) Astronomical Data Center. ADC data repository [http://tarantella.gsfc.nasa.gov/xml/], Greenbelt, MD

Shin DW, Jang HC, Jin HL (1998) BUS: an effective indexing and retrieval scheme in structured documents. In Proceedings of Digital Libraries '98. ACM, New York, pp 235–243

Shin DW (1999) Making XML documents searchable through the Web. In XML Developers Conference '99. Graphic Communications Association, Montreal, Canada, p 1

W3C (1998) Extensible Markup Language (XML) 1.0. W3C recommendation, REC-xml-19980210 [http://www.w3.org/TR/1998/REC-xml-19980210.html]

W3C (1999a) XSL Transformations (XSLT) version 1.0. W3C recommendation [http://www.w3c.org/TR/1999/REC-xslt-19991116.html]

W3C (1999b) XML Path Language (XPath) version 1.0. W3C recommendation [http://www.w3c.org/TR/1999/REC-xpath-19991116.html]

Correspondence and offprint requests to: Dongwook Shin, National Library of Medicine, 8600 Rockville Pike, Bethesda, MD 20894, USA. Email: dwshin@nlm.nih.gov