Integration of IR into an XML Database

Cong Yu
Department of EECS
University of Michigan
congy@eecs.umich.edu

Hong Qi
School of Information
University of Michigan
hqi@umich.edu

H. V. Jagadish
Department of EECS
University of Michigan
jag@eecs.umich.edu

ABSTRACT

Structure matching has been the focus and strength of standard XML querying. However, textual content is still an essential component of XML data. It is therefore important to extend the standard XML database engine to allow for "Information Retrieval" style queries, namely, "keyword" based retrieval and "result ranking". In this paper, we describe our effort in integrating information retrieval techniques into the Timber XML database system being developed at the University of Michigan, and our participation in the *IN*itiative for the *E*valuation of *X*ML Retrieval (INEX).

1 Introduction

With the growing popularity of XML, it is expected that more and more information will be stored and exchanged in XML format. Part of the information will be contained in the structure of the document. Another part, however, will be contained in a textual format within the elements (i.e, document components) of the XML documents. While boolean style querying is useful in some circumstances, there is a growing demand for querying both the textual information and the structure information in a non-boolean way. There are two general approaches to this problem. One is to start with a traditional IR system and augment it with the ability to recognize and extract the document structure. The other approach is to integrate IR facilities for querying textual content into a standard XML database engine, which handles structured queries well. We follow the second, database-oriented, approach, starting with Timber [13], a native XML database we have been developing.

There are four main challenges to this database-oriented approach. First, how to fit keyword based retrieval of the document components into the pipelined query evaluation of the database engine. Second, how to efficiently calculate the score of the matching elements to allow for future ranking of those elements. We developed `PhraseFinder` and `TermJoin` algorithms [3] to address both issues. The `PhraseFinder` algorithm uses a sort-merge based method to allow for pipelined retrieval of elements containing specified phrases (e.g., "information retrieval", instead of "information" and "retrieval"). The `TermJoin` algorithm is a stack-based algorithm that allows efficient retrieval of elements, at multiple granularities, that have a non-zero score according to a user-defined score function.

Third, how to aggregate the query results (i.e., a set of document components at different levels) such that users are not presented with redundant information, especially when they do not specify which type of elements to return (e.g., a content-only query). To address this so-called result redundancy issue, we have been working on the `Pick` algorithm which scrutinizes the result set according to a user-defined pick function and eliminates redundant elements using a stack based strategy.

Yet another main challenge in integrating IR into XML query is the specification of the query. We have devised a bulk algebra, TIX, for query *T*ext *I*n *X*ML, and several extensions to the XQuery language that give a framework on how IR style queries can be expressed at both the algebra level and the language level. Both TIX and the XQuery extension are out of the scope of this paper, interested readers are encouraged to take a look at [3].

The rest of the paper is organized as follows. Section 2 describes the Timber system and how it deals with structured queries. Section 3 describes the extensions to Timber that make the evaluation of IR style XML queries in Timber possible. In Section 4, we report our experience in using the Timber system to answer the set of INEX queries. Finally, we conclude in Section 5.

2 The Timber System

Timber [13] is a native database system currently being developed at the University of Michigan. The objective of the Timber system is to build

an efficient database engine for storing and querying XML data. It is based upon the TAX (*T*ree *A*lgebra in *X*ML) algebra [14] as its theoretical foundation for manipulating tree structures. Several access methods have been developed to retrieve the natively stored XML elements and a comprehensive pipelined query processing engine is implemented in the system to evaluate queries in the XML context.

The overall system architecture of Timber is illustrated in Figure 1. The whole system is built on top of the Shore object-oriented storage manager [20] (we are also developing another version that is built on top of the Berkeley DB backend store [7]), which is responsible for buffer management and concurrency control. The rest of Timber is composed of several components. XML documents are first parsed by the Data Parser to produce parse trees as inputs to the Data Manager. The Data Manager then transforms the nodes of those parse trees into an internal representation and stores them into the Storage Manager. Index Manager and Metadata Manager, as their name suggest, builds indices on the data and stores statistics about the data, respectively. At the heart of Timber is the Query Evaluator. It executes *evaluation plan*, which is produced by the Query Parser and optimized by the Query Optimizer, by interacting with the Data Manager and Index Manager. The details of the Timber system can be found in [13]. In particular, the attributes of an element are combined together and stored as a child element to the original element. Similarly, the textual content of an element is also represented as a child element to the original element. Therefore, nodes stored in Timber can be mainly classified into three types: element, text, and attribute, each has a slightly different format.

Structural queries can be efficiently processed by Timber. Each node in an XML document is represented by a triple *(startkey, endkey, level)*, where the *startkey* uniquely identifies the node in the database. In the case of multiple documents, the *startkey* of nodes in subsequent documents are incremented by an *offset* to make them unique in the entire database. A very important property of this coding scheme is that all the descendent nodes of a particular node n will have a *startkey* larger than $n_{startkey}$ and an *endkey* smaller than n_{endkey}. With this property, whether two nodes fit the ancestor/descendent or parent/child relationship can be determined in constant time by examining the two triples. It allows for efficient processing of structural joins (i.e., containment queries) using a stack based algorithm [2].

The Query Evaluator currently is able to process most of the XQuery expressed in the format of

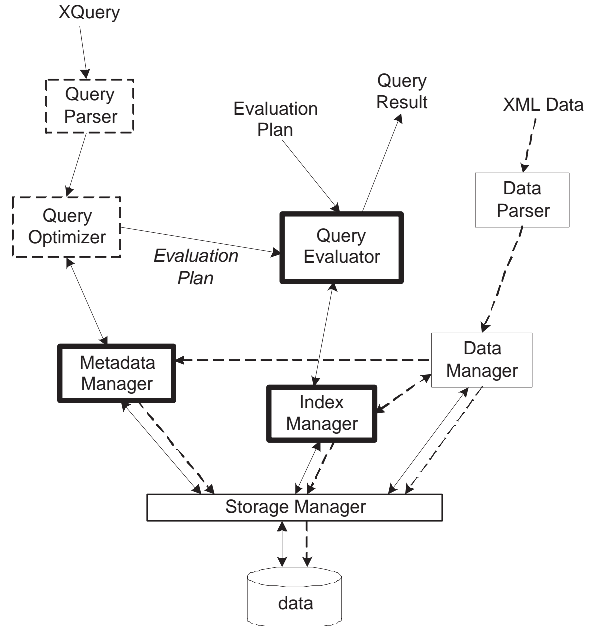

Figure 1: Timber System Architecture. Dashed lines represent loading data flow, solid lines represent retrieval data flow. Dashed rectangles represent components not used in INEX, solid rectangles represent components in Timber being used in INEX unmodified, bold solid rectangles represent components modified for integration of the IR extensions.

Timber Evaluation Plan, while the Query Parser and Query Optimizer can handle a smaller subset of the XQuery. In participating in the INEX, we have primarily used the *Evaluation Plan* interface instead of the XQuery interface because the XQuery interface was still under development at that time. However, in the future, we expect to be primarily using the XQuery interface so as to utilize the automatic query optimization provided by the Query Optimizer.

3 IR Extensions of Timber

To allow efficient processing of IR style query on the XML data, several components of the Timber system need to be extended. First, an IR-style inverted index is required to process keyword based search. Second, some extra information (e.g., how many words a text element contains) needs to be maintained by the Metadata Manager. Third, a score function needs to be integrated into the Query Evaluator to calculate relevance scores (i.e., return status value, *rsv*) to the matching elements. Fourth, an extra module is needed for eliminating redundant nodes in the final output set in the case when the user does not specify the type of elements to be returned. We describe the first two

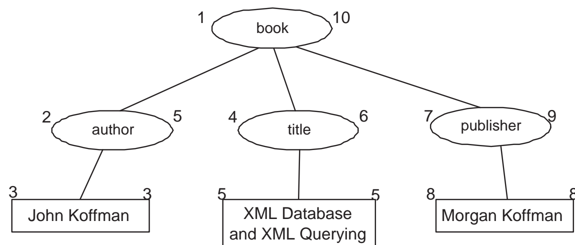

Figure 2: A Simple XML Document: ellipse indicate element nodes and rectangle indicate text nodes. The numbers on the shoulders of each node are the startkeys and endkeys respectively. The startkey and endkey for a text node are the same because it does not contain any child nodes.

keyword	(startkey, level, offset)
john	(3, 3, 0)
koffman	(3, 3, 1), (8, 3, 1)
xml	(5, 3, 0), (5, 3, 3)
database	(5, 3, 1)
query	(5, 3, 4)
morgan	(8, 3, 0)

Table 1: Sample Inverted Index Entries Based on Figure 2. Note that only startkey is needed since for text nodes, startkey is the same as the endkey.

extensions in this section and the latter two in Section 4.

3.1 Indexing INEX Data

Indices have been an integral part of Timber from the very beginning. Timber maintains several major types of indices. The most important ones include: 1) element tagname index, which maps a string s_1 to a set of element nodes (in the form of *(startkey, endkey, level)* triples) with tagname equal to s_1; 2) attribute name index, which maps a string s_2 to a set of element nodes that contain an attribute with the name equal to s_2; 3) attribute content index built on the element nodes with a specific attribute *attr*, which maps either a string s_3 or a number n_3 (floating-point number or integer number) to a set of element nodes that contain the *attr* attribute and have a value of s_3 or n_3 in *attr*; 4) element content index, which maps either a string s_4 or a number n_4 to a set of text nodes that have s_4 or n_4 as their value.

The aforementioned indices, however, are inadequate in supporting keyword based searches as required by IR style queries. An IR style query usually asks for document components related to a certain topic, which is frequently described as a set of terms (keywords) that, in the user's view, best capture the concept of the topic. Therefore, the results of an IR query are those elements that have the related terms in their textual content. The frequency and position of the term occurrences indicate the relevance of the element to the query.

To allow fast retrieval of elements that contain certain keywords, we extended the Timber Index Manager to include an inverted index on the text nodes. The index structure maps a word to the set of text nodes that contain the word. It also keeps track of the word offset in the textual content to allow matching of phrases (being used by `PhraseFinder`). Using the simple XML document in Figure 2 as an example, a total of six entries will be added to the inverted index, which are listed in Table 1.

A few strategies are employed to reduce the space requirement of the inverted index and to improve its accuracy. First, we compiled a list of most frequent used words in the INEX data set. Based on this list, we generated a list of 322 stop words for which we do not index, thereby reducing the index size significantly. Second, we do stemming of the words to index only the original form of the word (e.g., "query" instead of "querying"). The first stemming strategy we tried was the Porter's algorithm [16]. However, we found that Porter's algorithm is sometimes too aggressive (i.e., changing a word into its root instead of its original form). Instead, we decided to use WordNet's dictionary [22] to search for the original form of a word. This increases the time to build the inverted index, but it has the advantage of being more reliable than the Porter's algorithm.

3.2 Indexing Metadata

Another important extension is to the Timber Metadata Manager. As we will see in Section 4.2, for each node to be scored by the score function, not only the keyword occurrences in its textual content are needed, but also some extra statistics (metadata) related to the node in the context of the database. We have kept two main pieces of metadata information. First, the number of child nodes an element node has. Second, the total number of words a text node contains. Both are created by the Metadata Manager at the time the INEX data is loaded into the Timber. They can also be created after the loading in one pass over the entire database. We call them metadata indices to distinguish them from the normal indices that are maintained and accessed through the Index Manager. As a special variation to the first metadata index, we also maintain a separate index for <article> and <sec> element nodes that

is tuned for the INEX data. For an <article> node, we index how many <sec> nodes in its entire subtree, and for <sec> nodes, we index how many <p> and <p1> nodes in its subtree. This special variation is employed because we have discovered that <article>s, <sec>s, and <p>s are frequently the most reasonable return units in response to an INEX query. Having this special index can significantly reduce the time it takes to perform the redundancy elimination as described in Section 4.3.

3.3 Integration of Scoring and Redundancy Elimination into Query Evaluator

Scoring of each matched node is integrated into the query evaluation engine. Adopting a tree structured view toward XML document, the score function in our framework maintains a *localization* property: i.e., all the information needed by the score function in order to determine the score of a particular node is contained in the subtree rooted at that node or can be obtained via the Metadata Manager using one of the metadata indices. This allows the scoring of each node to be pipelined using the stack based `TermJoin` algorithm and therefore integrated into the evaluation engine (discussed in Section 4).

The coverage issue as highlighted in the INEX result assessment documentation [8] has two important aspects. First, there will be nodes covering only a subset of the information content being queried, so called *small coverage* or *partial coverage*. Second, there will be nodes covering information content not related to the query, so called *large coverage*. It is notable that the two aspects are orthogonal, i.e., a node can be covering only a subset of the requested information while having some information not related to the query. A node with a *small coverage* can be penalized by engineering the score function so that nodes with *complete coverage* or *full coverage* are assigned a higher score even though the absolute volume of information they contain is less. A node with a *large coverage* can be penalized by taking the size of the node into consideration in the score function. However, the introduction of structures in XML poses problems to this approach of attacking the *large coverage* problem. Imagine a query that matches a node with five child nodes, four of them are relatively small but not related to the query, only the relatively large child is related. Both the parent node and the related child node are likely to be returned to the user. However, the parent node should not because all the information it can give to the user is present in its child node. There are also cases where the parent node instead of the child nodes should be returned (see Section 4.3). We call this the result redundancy issue.

We utilize the pick function, which implements the `Pick` algorithm [3], to address the result redundancy issue. It is added as a module at the end of each query evaluation. The input to this module is a set of scored nodes. User defined criteria are employed by the module to select those nodes at the appropriate granularity level. Metadata indices are also being used to help remove the redundancy. A default selection criterion is always provided in case user does not provide one. The output of the module is a set of nodes at, hopefully, the right granularity level.

The detailed description of both score function and pick function can be found in [3] and in the following section.

4 IR Query Evaluation

There are three phases in processing each INEX topic. First, the topic is translated into an *evaluation plan* that the Query Evaluator can understand. Second, the plan is executed to produce a set of nodes, each with a score indicating how relevant it is to the query. Third, in the case of a content-only query, a final result redundancy elimination procedure is performed. The final result can then be sorted and a certain number of the top results are returned to the user. Throughout this section, we will use the two query topics in Figure 3 as our running examples.

4.1 Topic Translation

The topic translation accomplishes two important tasks: one is translating the XPath expression in the original topic into *evaluation plan*; the other is categorizing keywords into several classes to be used by the score function. The first task can be accomplished relatively easily because Timber already supports most of the XQuery, a superset of XPath. Essentially, each "/" is translated into a parent/child join, each "//" is translated into an ancestor/descendent join, and each <cw>/<ce> pair is translated into a term join. Figure 4 shows the *evaluation plan* of Topic 12 (a content-and-structure query) in its tree format. The ellipse nodes labeled with tag names are retrieved via the *element tagname* index. The rectangle nodes represent text nodes retrieved via the *inverted index*. The content in those nodes dictates how they should be scored by the score function and is explained later in this section. Finally, the edges between two nodes indicate the join algorithms being used to fetch them, including parent/child join

```
<INEX-Topic topic-id="12" query-type="CAS" ct-no="069">
   <Title>
      <te>article/bdy/sec</te>
      <cw>2001 2002</cw><ce>article//pdt/yr</ce>
      <cw>internet search engine</cw>
      <ce>article/bdy/sec</ce>
   </Title>
   <Description>
      Retrieve sections of articles published in 2001
      or 2002 on the topic of internet search engines.
   </Description>
   <Narrative>
      To be relevant, the article should talk about
      the current status of internet search engines,
      problems associated with current technologies,
      and future developments.
   </Narrative>
   <Keywords>
      internet search engine information retrieval
   </Keywords>
</INEX-Topic>

<INEX-Topic topic-id="31" query-type="CO" ct-no="003">
   <Title>
      <cw>computational biology</cw>
   </Title>
   <Description>
      Challenges that arise, and approaches being
      explored, in the interdisciplinary field of
      computational biology.
   </Description>
   <Narrative>
      To be relevant, a document/component must either
      talk in general terms about the opportunities
      at the intersection of computer science and biology,
      or describe a particular problem and the ways it is
      being attacked.
   </Narrative>
   <Keywords>
      computational biology, bioinformatics, genome,
      genomics, proteomics, sequencing, protein folding
   </Keywords>
</INEX-Topic>
```

Figure 3: Two Example INEX Query Topics

(*PC Join*), ancestor/descendant join (*AD Join*), and the IR specific *Term Join*. The translation of a content-only query (e.g., Topic 31) to the *evaluation plan* is also easy. As shown in Figure 4, it involves term-joining matching text nodes with all their ancestor element nodes regardless of the tag name. The actual *evaluation plan* is in text format and is omitted here.

The second task is to separate the set of keywords provided in the <Keywords> part of the topic into three categories: REQ, HIGH, and LOW. A keyword appears in the REQ category if it is listed in the <Title>/<cw> part of the original topic. A keyword appears in the HIGH category if: 1) it is not in the REQ category and 2) it appears in the <Description> or <Narrative> part of the topic. Finally, a keyword is in the LOW category if it appears only in the <Keywords> part. Keywords falling into different categories will have different weights in contributing to the overall score of the element. More importantly, a node with all the REQ keywords is always assigned a higher score than one missing some of the REQ keywords, regardless of the other two categories. This means a node with *full coverage* of the information is always ranked higher than a node with *partial coverage*. In addition to categorizing keywords, we also try to identify keyword phrases by scanning through the <Description> and <Narrative> parts of the topic. A keyword phrase is identified if two or more consecutive words occurring in the <Keywords> part also occur in <Description> and <Narrative> parts in the same consecutive order. For example, the phrase "internet search engine" can be identified in Topic 12. It is worth mentioning that in some topics, the author specifies the phrases in certain format (e.g, Topic 31), which means this extra phrase identification step is not required in all topics. We do find that phrase identification has a very significant impact on the result accuracy, which suggests that some better mechanism for specifying or identifying phrases in the set of provided keywords is worth further investigation. The search engine *www.alltheweb.com* has made some efforts in this direction. It is also worth noting that some <cw>s are actually exact boolean matches rather than keyword based matches (e.g., Topic 12 specifies that the publication year of the article must be 2001 or 2002). However, there seems to be no easy way for the topic translation script to automatically recognize this. We therefore may have to manually notify the score function of this. The result of this keyword categorization is the content in the rectangle text nodes as shown in Figure 4.

The *evaluation plan* is then provided to the Query Evaluator for execution and the result of keyword categorization is supplied to the score function inside the Query Evaluator, which uses it to calculate scores for each matched node.

4.2 Score Generation

Score generation is accomplished by the score function inside the Query Evaluator. For INEX, we use a default score function based on the following formula:

$$score = \sum_{j=req,high,low} \left(\frac{W_j}{N_j} \sum_{i=1}^{N_j} \frac{\log N_{keyword_i}}{size_{node}}\right) \quad (1)$$

where W_j is the weight assigned to one of the three categories, N_j is the total number of keywords (a phrase is counted as one keyword) in that category, $N_{keyword_i}$ is the number of occurrences of a certain keyword in the current node, and finally, $size_{node}$ is the total number of words in the current node. For INEX, we have used

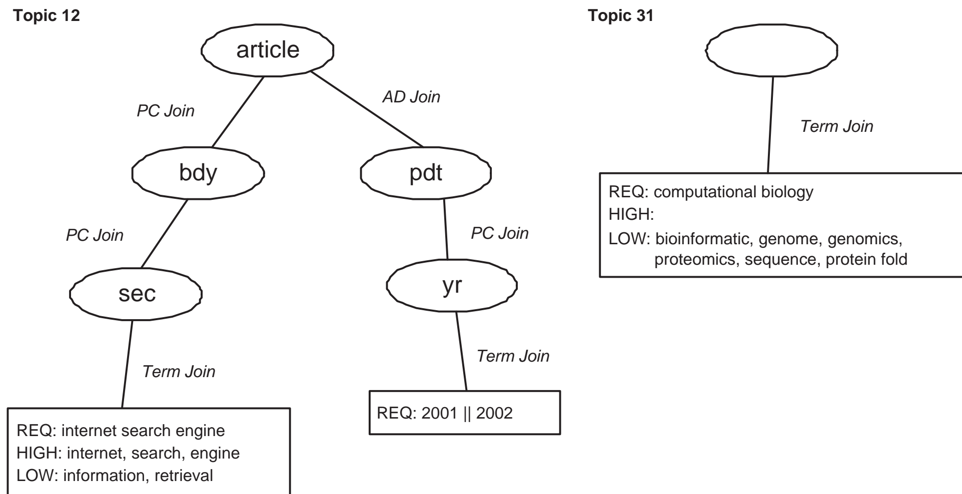

Figure 4: The *evaluation plan* of Topic 12 and Topic 31 in Tree Format.

$W_{req} = 0.6$, $W_{high} = 0.25$, and $W_{low} = 0.15$. As mentioned before, for nodes with all the REQ keywords, we add to it a constant that is large enough to make its score higher than any node without all the REQ keywords.

For a content-and-structure query, only the element nodes satisfying the structure requirement are processed by the score function. For a content-only query, all the element nodes that are ancestors of a text node, containing at least one of the keywords, are processed. For certain content-and-structure queries with, for example, <au> as the target element, we recognize that the real intention of the user is to rank <article>s while retrieving <au>s of the matched <article>s. In those cases, the <article> elements are processed by the score function instead of the <au> elements.

4.3 Redundancy Elimination

For content-only queries, the set of results produced by the score function can contain many redundant information. For example, it is possible that an <article> and all its five <sec>s have high scores. But the user should really be only presented with the <article> since that is the one that contains all the information. The example in Section 3.3 also illustrates the case where a child component, rather than the parent component, should be returned.

This redundancy elimination is accomplished by the pick function. For INEX, we have employed a special pick function that operates only on three major types of element nodes: <article>, <sec>, and <p>. The basic idea is as follow. To decide whether to return an <article> or a <sec> underneath it, we check how many <sec>s under that <article> are relevant (a node is considered relevant if it has a score that ranks it in top 500 when all the nodes being processed by the score function are considered). If above a certain percentage (we default it to 50%) of the <sec>s (among all the <sec>s underneath that <article>) are relevant, the <article> is picked and the <sec>s are discarded. Otherwise, the individual <sec>s are returned. Nodes of other types fall into two categories: one that is underneath an <article>, the other that is not. Nodes in the first category are discarded since the information contained in them is captured in one of the above three types. Nodes in the second category are kept because the information within them can not be captured in any of the above three types.

After redundancy elimination, all the remaining nodes are sorted and the top 100 are then returned back to the user.

4.4 Performance

We briefly discuss the performance of our system in terms of two measurements: storage space and querying time.

The entire INEX data occupies about 5GB (a roughly ten-time increase from the original data) disk space to accommodate both the raw data and the extra structural information needed (e.g., *startkey*, *endkey*, etc.). All the auxiliary indices (e.g., element tag index) are quite small. The only exception is the inverted index, which is considerably large compared to the other indices. The problem is made worse due to the fact that we are using GiST [11] as our physical level index manager, which leaves us no control over how things are organized on the disk. We are currently investigating ways to control the size of the inverted index without loss of efficiency.

Once the INEX topic gets translated into the *evaluation plan*, its execution time depends on how frequent the keywords are in the data set and how many structure constraints are in the query. The complexity of `TermJoin` is $O(\sum(T_i))$, where T_i is the number of how many times keyword T_i occurs in the data set. Therefore, the more frequent the keywords are, the longer it takes to evaluate the query. Structural constraints also plays an important role here because the Query Evaluator can quickly discard elements that do not satisfy the structural condition without trying to score them. On average, content-and-structure queries can be evaluated within a few seconds of CPU time. While content-only queries can take from several seconds to over a minute to finish.

Another component of querying time is the time it takes to translate the INEX topic into *evaluation plan*. As discussed in Section 4.1, although the topic translation is automated, the ambiguities in the topic specification mandate some manual work to ensure the resulting *evaluation plan* can be correctly executed by Timber. This step, which involves reading through and understanding both the topic and the plan, usually takes a few minutes.

5 Conclusion

In this paper, we described our participation in INEX. In particular, we described how we have extended Timber, a native XML database system, to query structured text in the format of XML.

The official assessment results from INEX indicate that, when equipped with IR extensions, Timber performs quite well in querying XML data (with regard to the topics whose assessments are finished). We believe the success comes from two aspects. First, as an XML database engine, Timber is able to handle structure constraints with ease. For content-and-structure queries, Timber can significantly reduce the number of document components to be scored based on the structure conditions. Second, the integration of the score function and pick function into the Query Evaluator allows Timber to efficiently assess a component based on keywords (score function) and structural containment (pick function), which makes it suitable to process IR style non-boolean queries.

ACKNOWLEDGEMENT

This work is supported in part by National Science Foundation under Grant No. IIS-0208852. We would like to thank members of the Timber research group at the University of Michigan for their technical assistance and helpful advice.

References

[1] Rakesh Agrawal and Edward L. Wimmers. A framework for expressing and combining preferences. In *International Conference on Management of Data (SIGMOD)*, 2000.

[2] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: A primitive for efficient XML query pattern matching. In *International Conference on Data Engineering (ICDE)*, Heidelberg, Germany, April 2001.

[3] Shurug Al-Khalifa, Cong Yu, and H. V. Jagadish. Querying structured text in an XML database. In *International Conference on Management of Data (SIGMOD)*, San Diego, CA, June 2003.

[4] Nicolas Bruno, Luis Gravano, and Amelie Marian. Evaluating top-k queries over web-accessible databases. In *International Conference on Data Engineering (ICDE)*, 2002.

[5] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *International Conference on Management of Data (SIGMOD)*, 2002.

[6] William Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *International Conference on Management of Data (SIGMOD)*, 1998.

[7] Berkeley DB. http://www.sleepycat.com/.

[8] DELOS. Initiative for the evaluation of XML retrieval. http://qmir.dcs.qmw.ac.uk/inex/.

[9] Norbert Fuhr and Kai Großjohann. XIRQL: A query language for information retrieval in XML documents. In *International Conference on Information Retrieval (SIGIR)*, 2001.

[10] Norbert Fuhr and Thomas Rölleke. A probabilistic relational algebra for the integration of information retrieval and database system. *ACM Transactions on Information Systems (TOIS)*, 15(1), January 1997.

[11] GiST. http://gist.cs.berkeley.edu/.

[12] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A system for the efficient execution of multiparametric ranked queries. In *International Conference on Management of Data (SIGMOD)*, 2001.

[13] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V.S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, NuweeWiwatwattana, Yuqing Wu, and Cong Yu. TIMBER: A native XML database. *The VLDB Journal*, 11(4):274–291, 2002.

[14] H. V. Jagadish, Laks V.S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A tree algebra for XML. In *International Workshop on Database Programming Languages (DBPL)*, Marino, Italy, September 2001.

[15] Andrew Nierman and H. V. Jagadish. ProTDB: Probabilistic data in XML. In *Very Large Data Bases (VLDB) Conference*, Hong Kong, China, August 2002.

[16] M.F. Porter. An algorithm for suffix stripping. *Program*, 14(3), 1980.

[17] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[18] Torsten Schlieder and Holger Meuss. Result ranking for structured queries against XML documents. In *DELOS Workshop on Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[19] Albrecht Schmidt, Martin Kersten, and Menzo Windhouwer. Querying XML documents made easy: Nearest concept queries. In *International Conference on Extending Database Technology (EDBT)*, 2001.

[20] Shore. http://www.cs.wisc.edu/shore/.

[21] Anja Theobald and Gerhard Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In *International Conference on Extending Database Technology (EDBT)*, 2002.

[22] WordNet. An electronic lexical database. http://www.cogsci.princeton.edu/ wn/.

[23] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *International Conference on Management of Data (SIGMOD)*, 2001.