International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:3, 2008

Data Extraction of XML Files using Searching and Indexing Techniques

Sushma Satpute, Vaishali Katkar, and Nilesh Sahare

***Abstract*—**XML files contain data which is in well formatted manner. By studying the format or semantics of the grammar it will be helpful for fast retrieval of the data. There are many algorithms which describes about searching the data from XML files. There are no. of approaches which uses data structure or are related to the contents of the document. In these cases user must know about the structure of the document and information retrieval techniques using NLPs is related to content of the document. Hence the result may be irrelevant or not so successful and may take more time to search.. This paper presents fast XML retrieval techniques by using new indexing technique and the concept of RXML. When indexing an XML document, the system takes into account both the document content and the document structure and assigns the value to each tag from file. To query the system, a user is not constrained about fixed format of query.

***Keywords*—**XML Retrieval, Indexed Search, Information Retrieval.

I. Introduction

NOW a day there is huge amount of data from all fields is available on internet. By using simple keyword technique user may get more time to retrieve the data, as data is available in a huge plain text. However, web search engines index mainly plain documents as texts and HTML pages. When data is structured, or semi structured, by studying the structure of data we can get the nearly exact information in less amount of time. The semantics of the data is conveyed by content and structure of the data. Thus, it appears important to index the structure in order to capture all the semantic of a document. An exact matching paradigm supported by XML query languages such as W3C's XPath or XQuery has widely proved his effectiveness. But, XML query languages are very complex for a naïve user and require a prior knowledge about the structure of the documents searched. Such knowledge is hardly available in Windows and Web environment.

Hence indexing XML documents must be for: It is well indexing the structure and the content of XML documents, in a way which preserves the document semantic and give the search the document in short amount of time, and allows a simplest user query language. This is done by our system.

Some approaches we have cited are mainly for plain text retrieval and database retrieval oriented. For database approach they have mainly considered query language, for such XQuery [14] which is selected as the basis for an official W3C query language for XML. We can also cite XPath [6] which is the ancestor of XQuery, XQL [4] etc... A comparative study of some query languages is published in [15]. IR-oriented approaches use techniques of IR to index and search XML documents. Some of these approaches are an adaptation of traditional IR-models to XML search [15] like the Boolean and the probabilistic models, the vector space model is extended also to search XML documents [9]. Other approaches, uses adaptations of techniques like `tf-idf` to XML data, like for example, XSearch [10] and XRank [11].

A. About the System

While designing the system we are mainly concerned about the time required to access the data. As XML is a File based accessing scheme so while accessing by multiple applications at a time some integrity problem may occur .We have overcome this problem by creating a new type of XML Management Server. Through this we can make XML File access as connection oriented. A query is submitted to the system and a list of documents is searched in public folder and displayed along with weight of the document in return. Hence the user can discover the context of the information returned, this, helps user to assess the relevance of a result.

The search engine is made of two main modules.
1) A Document Indexer and
2) A Query evaluator.

The document goes through the document parser which analyses its structure and contents perform indexing on it, while the query from user goes through query evaluator. In query evaluator the meaningful words are extracted or matched from query and they are searched in document indexer. If they are found the result is displayed according to relevancy to original document.

Diagrammatically the system can be shown in following Fig. 1.

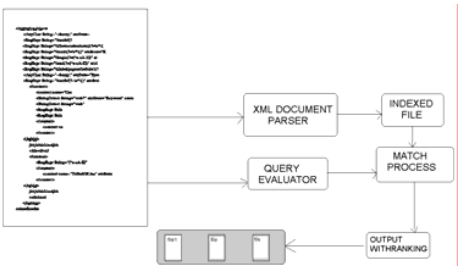


Fig. 1 Diagrammatic Representation of Search Engine

II. Arranging the Index

Indexing the document: We need three indexes to store the value associated with each node,

First is the file indexer to keep track of each file that will be file-id.

Second is weight of the node which is calculated according to occurrences of the tag.

Third is the distance of the node from the root node.

To index a document, our system performs operations, which need three index structures:

- fmanager table archiving the files indexed.
- nlist table keeping the structure of documents archived.
- wlist table keeping every word appearing in documents archived.

We choose to store the index in a MySql database.

Indexing Document Structure

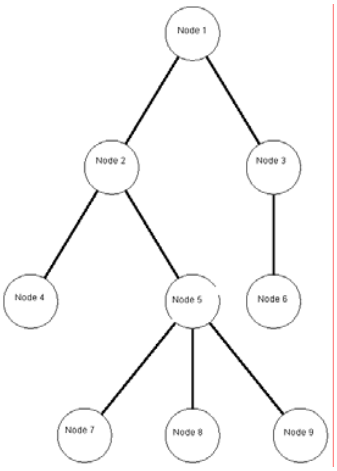
For now, we expose the simple case of a document without id references. Since the storing structures of MySql are tables that have a fixed number of columns, the idea is to transform the tree into a table, while allowing navigation in both directions. Each node tree represents an element or an attribute. They are stored in a table (*nodelist*) containing the following parameters:

• *name* is the name of the element or attribute,
• *idfile* is the ID of the document,
• *idnode* is the unique ID (in the entire tree) of the node itself,
• *child1* is the ID of the first child of the current node,
• *childcount* is the number of children of the node (the number of attributes plus the number of elements),
• *parent* is the ID of the parent node

The principle is quite simple: each element or attribute is represented by a node. Each node is assigned an identifier ID called idnode. Identifiers are stored by using breadth-first technique, so, every child that is *"neighbor"* is adjacent in the *nodelist*. To refer to its children, a node must specify the idnode of its first child and the child number. In the Fig. 2 we can see an example of a tree and its associated *nodelist*. Particular values are attributed to some parameters:

• The root node has always an idnode set to 0.

• The leaves have a childcount equal to 0. Then, the value of child1 is :
– 1 if the node is an element.
– 0 for nodes representing attributes.



(a) Tree structure of Document

ID	Node	Parent	ChildCount	Child1	Weight
0	Node1	0	2	1	7
1	Node2	1	2	3	5
2	Node3	1	1	5	8
3	Node4	2	0	/	4
4	Node5	2	3	6	6
5	Node6	3	0	/	9
6	Node7	5	0	/	1
7	Node8	5	0	/	2
8	Node9	5	0	/	3

(b) Table structure

Fig. 2 Example of a tree structure and its associated *nodelist*

III. Query Evaluator

To evaluate a query, the system first transforms it in an adequate structure, this step is called here *query analysis*. Subsequently, the index structure is searched and the relevant documents are listed.

A. Query Analysis

For this search engine, a query is a set of keywords given by the user and separated by operators. We can say operators are words which join the two words; they may be Boolean or commonly used English language words. The matching documents must contain the words matching the query keywords. The operators include the common Boolean operators or commonly used English language words .They are described below: ( ) The parentheses classically change the order used to resolve the expression. AND this performs the usual Boolean *and* operator. It is equivalent to the '+' or the '&' symbol or when no symbol is specified. OR This performs the usual Boolean *or* operator. It is equivalent to the '|' symbol. Query analysis consists of parsing the queries and transforming it into a tree where nodes are Operators and leaves contain the keywords.

B. Searching the Index

Searching the index is done in two steps.

a) First, the system localizes nodes containing the query keywords, secondly, it eliminates those nodes or documents which don't match the query specifications expressed by the operators. Before detailing these two steps, we initially present the structures used to store the results. For each query keyword, our system assigns a structure, called Resultquery. It is a set of NodeSet, one by document in which the keyword occurs. Each NodeSet contains a unique ID (called idfile) and a set of Score. Each Score structure has got an ID (called idnode) corresponding to the node where the keyword occurs and two vectors *cnttype* and *proxim* described below.

Values in *cnttype* and *proxim* will be used to rank the documents. *cnttype* is a vector containing the types of the keyword occurrences. Each element of this vector takes one of the following values:

- 1 if the occurrence of the keyword is a tag name,
- 2 if the occurrence of the keyword is an attribute name,
- 3 if the occurrence of the keyword is a tag value,
- 4 if the occurrence of the keyword is an attribute value. Note that these values have not any other role than representing a term type. *proxim* is a vector that keeps the distance of the keyword occurrence, in term of node nesting, with the root node. The Fig. 3 schematizes these structures. In all vectors, the structures presented above are always sorted according to the ID of the item to which it refers. The node localization step consists of filling the ResultQuery structure presented above for each query key. To perform this, the system creates the no of nodes as many as the total no of keywords present in the document. And for each NodeSet, the system creates as many Scores as nodes in the corresponding document which contains the keyword.

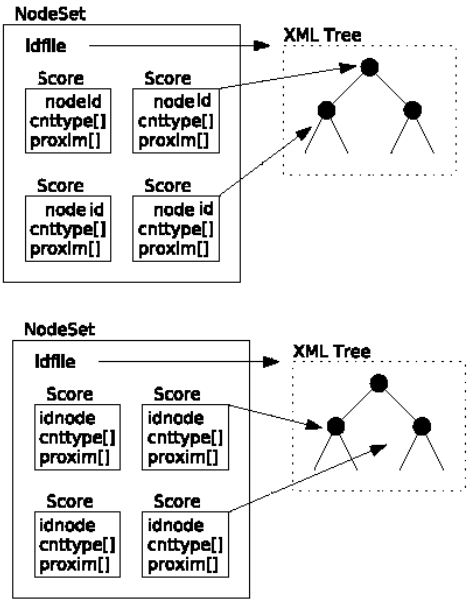


Fig. 3 Structure used to evaluate query

Moreover, for each ancestor of those nodes, a Score is created. The Score structures of the ancestors have their value proxim incremented for each additional level. To determine the documents (NodeSets) matching the query, some rules are applied to merge the Resultquery corresponding to query keywords. These rules are operations on sets of NodeSets and sets of Scores. These operations are intersection, union. The way to handle the ranking parameters is explained in the next section. The intersection is used to perform an AND operator, the union, to perform an OR operation. Note that the intersection returns the first common ancestor of each pair of operand nodes. Therefore, if a document contains two keywords, the operator AND between these keywords returns at least a Score referring to the root node. For operators which involve document structure, we use an operation a bit more complex.

1) Each fileid appears in Resultquery of both operands. In other words, that simply means that the keywords must appear in the same document.

2) These such NodeSet must contain the idnode that have a proxim equal to 0 in the left operand and greater than 0 in the right operand where:

1) As previous, each idfile appears in Resultquery of both operands.

2) These such NodeSet must contain the idnode that have a proxim equal to 0 in both operands. Thus, the query tree is traversable from leaves to root and Resultquery are merged accordingly to rules associated to the operators. When reaching the root, only one Resultquery remains and it contains NodeSets corresponding to the relevant documents. These NodeSets contain in turn Scores corresponding to the relevant fragments (nodes) in the document.

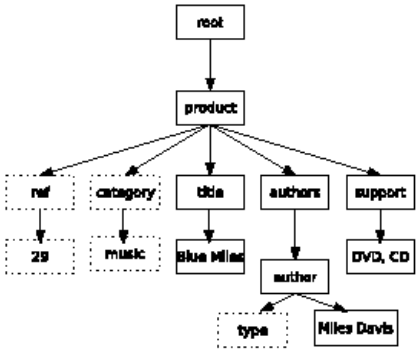
3) Example of query evaluation: Consider the following mini database containing two documents.

<root>
<Product ref="29" category="music">
<Title>
Blue Miles
</Title>
<Authors>
<Author type="artist">
Miles Davis
</Author>
</Authors>
<Support>
DVD, CD
</Support>
</Product>
</root>

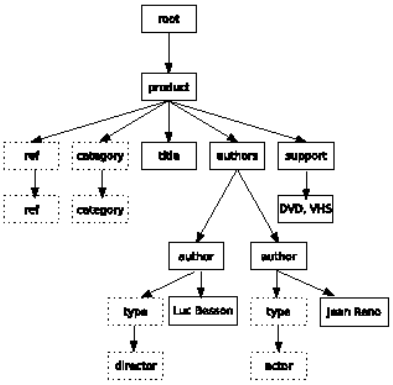
The second document is
<root>
<Product ref="36" category="movie">
<Title>
The Big Blue
</Title>
<Authors>
<Author type="director">

Luc Besson
</Author>
<Author type="actor">
Jean Reno
</Author>
</Authors>
<Support>
DVD, VHS
</Support>
</Product>
</root>

The XML-graphs associated with these documents are depicted on fig. D. Let us consider the query : "*blue AND title AND cd"* represented in D(b). First, three Resultquery are constructed for the three keywords contained in the query, namely," *blue*,*title* and *cd*." Then, the two Resultquery corresponding to first keywords blue and title are merged into unique query result which in turn merge with third query result i.e. query.



(a) XML-graph associated with the document doc_0



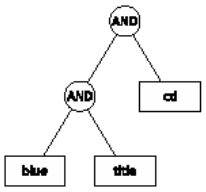
(b) XML-graph associated with the document doc_1

Fig. 4 XML Graphs

Result of the query tree resulted.

```
blue AND title AND cd
```

(a) Query



(b) Query tree associated

Fig. 5 Query Tree Generation

Evaluation of the query tree is as shown in figure below:

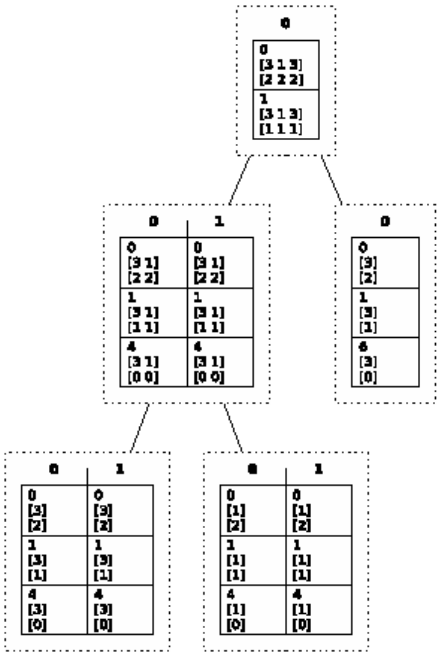


Fig. 6 Query Tree Evaluation

C. Assigning the Rank and Final Results

The nodes are ranked according to the relevance of the document to the node. Relevance of a fragment to the query depends on several factors listed below.

• First, the number of occurrences of query keywords in the underlying fragment. This measure is usually used in information retrieval to determine how well a term describes a document (in our case a fragment).

• Secondly, the position of query keywords within the fragment (tag name, tag value, attribute name, attribute value). The impact of the keyword type on the fragment relevance is not trivial. Intuitively, we suppose that words in an attribute value are more significant than those in a tag content like shows the following example. Let us consider the two parts of XML documents:

<book>
<title = "Reigen">
<author = "Schnitzler">
<summary>

… the scene that confronts
Emma and Alfred, the young man
who cite "Le rouge et le Noir"
of Stendhal to proof his love...
</summary>

</book>

<book>
<title = "Le Rouge et le Noir">
<author = "Stendhal">
<summary>
...
</summary>
</book>

Suppose that a user submits the query:

Stendhal AND Rouge

It is more relevant to give a higher rank to the second document since Stendhal and Rouge are attribute values that are more structured words than the words simply present in a tag content.

• Thirdly, the distance between query keywords (in term of nodes) within documents. Like in traditional information retrieval, we think that keyword proximity in a document enhances the relevance judgment of this document. Suppose for example that a user who is looking for *museums* in *Brussels* submits the query" museums AND Brussels". If retrieved fragments are ranked accordingly to the distance between *museums* and *Brussels* the fragment (a) in figure 7 would be ranked higher than the fragment (b)
in the same figure.

• Fourth, the hierarchical position between query keywords. We are considering the semantic of the document which relate parent and child elements.

On that basis, the element (a) in Fig. 7 would be ranked higher than the element (b) in the same Fig. 7.

• Fifth, the relevance of the document which the element belongs.

• Finally, The specificity of an element.

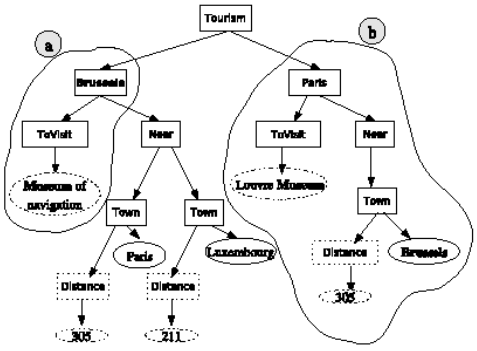


Fig. 7 Keyword Proximity

IV. Implementation of an Indexing and Querying Method Over XML Document

Let us consider the example of an XML document which contains the records of employees.

```
<?xml version="1.0" standalone="yes"?>
<company>
<Employees>
 <Employee1>
  <Eid>PR100</Eid>
  <Ename>NILESH</Ename>
  <EDesig>PROGRAMMER</EDesig>
 </Employee>
 <Employee2>
  <Eid>TL101</Eid>
  <Ename>VAISHALI</Ename>
  <EDesig>TEAMLEADER</EDesig>
 </Employee>
 <Employee3>
  <Eid>SA102</Eid>
  <Ename>PRASHANT</Ename>
  <EDesig>TEAMLEADER</EDesig>
 </Employee>
 <Employee4>
  <Eid>TL103</Eid>
  <Ename>AMOL</Ename>
  <EDesig>TEAMLEADER</EDesig>
 </Employee>
 <Employee>
  <Eid>PR</Eid>
  <Ename>SHASHANK</Ename>
  <EDesig>PROGRAMMER</EDesig>
 </Employee>
 <Employee5>
  <Eid>SA104</Eid>
  <Ename>HEMANT</Ename>
  <EDesig>SYSTEM ANALYST</EDesig>
 </Employee>
 <Employee>
  <Eid>SA105</Eid>
  <Ename>BIJOY</Ename>
  <EDesig>SYSTEM ANLYST</EDesig>
 </Employee>
 <Employee>
  <Eid>PR107</Eid>
  <Ename>ATUL</Ename>
  <EDesig>PROGRAMMER</EDesig>
 </Employee>
 <Employee>
  <Eid>SA108</Eid>
  <Ename>VEENA</Ename>
  <EDesig>SYSTEM ANALYST</EDesig>
 </Employee>
 <Employee>
  <Eid>PR109</Eid>
  <Ename>HEEMA</Ename>
  <EDesig>PROGRAMMER</EDesig>
 </Employee>
```

```
<Employee>
  <Eid>PR110</Eid>
  <Ename>AJIT</Ename>
  <EDesig>PROGRAMMER</EDesig>
 </Employee>
 <Employee>
  <Eid>pr12</Eid>
  <Ename>pqr</Ename>
  <EDesig>program</EDesig>
 </Employee>
</Employees>
```

Here to build the query we are taking the actual values as a query and searching it, first in tags and then matching to attributes and then value, as shown in following Fig. 8.

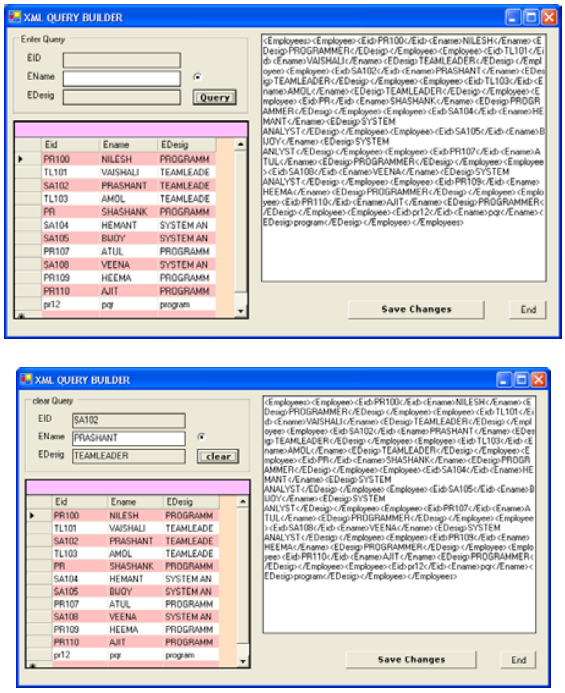


Fig. 8 Implementation of an indexing and querying method over XML document

In above program, we are able to add new values at run time. We also view the XML Document at the same time. Updations can be made runtime and these changes are directly stored in XML file and are visible runtime. And for this implementation we require R-XML API,s. As shown in the Fig 9. We need to develop following modules for making the application connection oriented environment and to create proper index between different XML files.

1. Client module
    a. Client Query Processor
    b. Client Network Model
2. Server module
    a. Server Network Model
    b. Client Query Analyzer
    c. Relation Manager Functions

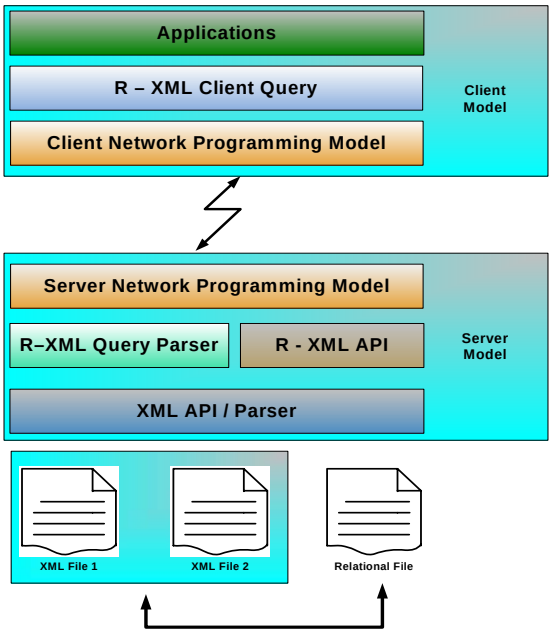


Fig. 9 R-XML Architecture

V. Conclusion

We have presented the design and naive technology for indexing. The distinctive feature of our indexing method is that it takes into account the keyword type (tag, content etc...) and keywords proximity along with keyword weight into XML documents. Regarding our querying method, it allows both complex and simple system querying. So, an ordinary user can submit a list of keywords when a more experienced user can submit complex queries to express structure constraints for example. Moreover, we have presented the elements which we think necessary to arrive at an effective ranking model for XML fragments. Like it said above, we are currently working on use of this technique on the distributed system.

References

[1] M. F. Porter, “An algorithm for suffix stripping,” *Program*, vol. 14, no. 3, pp. 130–137, 1980.
[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, “The lorel query language for semistructured data,” *JODL*, vol. 1, no. 1, pp. 68–88, April 1997.
[3] S. Brin and L. Page, “The anatomy of a large-scale hypertextual Web search engine,” *Computer Networks and ISDN Systems*, vol. 30, no. 1–7, pp. 107–117, 1998. [Online]. Available: citeseer.ist.psu.edu/brin98anatomy.html
[4] D. S. J. Robie, J. Lapp, “Xml query language (xql),” *QL’98 The Query Languages Workshop*, 1998, www.w3.org/TandS/QL/QL98/pp/xql.html.
[5] Xml path language (xpath) version 1.0,” Tech. Rep., November 1999, http://www.w3.org/TR/xpath.
[6] A. Bonifati and S. Ceri, “Comparative analysis of five XML query languages,” *SIGMOD Record*, vol. 29, no. 1, pp. 68–79, 2000. [Online]. Available: citeseer.ist.psu.edu/article/bonifati00comparative.html
[7] N. Fuhr and K. Grosjohann, “XIRQL: A query language for information retrieval in XML documents,” in *Research and Development in*

Information Retrieval, 2001, pp. 172–180. [Online]. Available: citeseer.ist.psu.edu/fuhr01xirql.html

[8] A. Theobald and G. Weikum, "The index-based xxl search engine for querying xml data with relevance ranking," in *EDBT '02: Proceedings of the 8th International Conference on Extending Database Technology*. London, UK: Springer-Verlag, 2002, pp. 477–495.

[9] D. Carmel, Y. Maarek, Y. Mass, N. Efraty, and G. Landau, "An Extension of the Vector Space Model for Querying XML documents via XML fragments," in *ACM SIGIR 2002 Workshop on XML and Information Retrieval, Tampere, Finland*, august 2002.

[10] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "XSearch : A Semantic Search Engine for XML ," in *29th VLDB Conference, berlin, Germany*, 2003, http://www.vldb.org/conf/2003/papers/S03P02.pdf.

[11] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "Xrank: Ranked keyword search over xml documents," 2003. [Online]. Available: citeseer.ist.psu.edu/guo03xrank.html

[12] H. Meyer, I. Bruder, G. Weber, and A. Heuer, "The xircus search engine," 2003. [Online]. Available: citeseer.ist.psu.edu/meyer03xircus. Html

[13] P. Francq, "Collaborative and structured search: an integrated approach for sharing documents among users," Ph.D. dissertation, Universit´e libre de Bruxelles, June 2003.

[14] W. W. W. Consortium, "Xquery 1.0: an xml query language," Tech. Rep., November 2003, http://www.w3.org/TR/xquery.

[15] K. Sauvagnat and M. Boughanem, "XFIRM: A Flexible Information Retrieval Model for Indexing and Searching XML documents," in *ECIR (European Conference on Information Retrieval)- Proceedings volume 2 (Poster Abstracts) , Sunderland, UK*. - Edited by Michael P. Oakes,5-7 avril 2004, pp. 17–18.