An Extension of the Vector Space Model for Querying XML Documents via XML Fragments[1]

David Carmel*, Nadav Efraty, Gad M. Landau**, Yoelle S. Maarek*, Yosi Mass***
*IBM Research Lab in Haifa
Haifa 31905, ISRAEL
** Computer Science Dept
Haifa University, Haifa, ISRAEL

Abstract

To date, most of the work on XML query and search has stemmed from the document management and database communities and from the information needs of business applications, as evidenced by existing XML query languages such as W3C's XQuery, which is strongly inspired by SQL. We propose here to extend the realm of XML by supporting the information needs of users wishing to query XML collections in a flexible way without knowing much about the documents structure. Rather than inventing a new query language, we suggest to query XML documents via pieces of XML documents or "XML fragments" of the same nature as the documents that are queried. We then present an extension of the vector space model for searching XML collections and ranking search results by relevance

1 Motivation

XML (the eXtensible Markup Language) [1] can be used not only for exchanging data in business-to-business applications, but also for representing any kind of free-text documents, especially narrative documents such as articles, business reports, and naturally Web pages. Most of the current XML query languages, *e.g.*, XQuery [2], and systems that embody them, are geared towards structure and data, and typically expect "binary answers" to very specific queries. While this is appropriate for XML documents that data exchange between applications, this is less so for discovering of narrative documents. As observed by [3], an overwhelming majority of the required features in the W3C proposal for an XML Query Languages are "data-centric", and only one is "document-centric", *i.e.,* dedicated to processing simple text conditions.

The need for document centric approaches has been already identified in the IR community, [4]. More recently the XQuery working group has been conducting debates in order to investigate how to add full-text search features and more particularly ranking capabilities in XQuery. In parallel, a variety of systems that support a document-centric approach have been proposed, [3,5]. While all of these efforts represent a significant step in this direction, they still focus on giving control to applications as well as users. We discuss below a simplified approach for concentrating exclusively on the information needs of users rather than the very specific control and management needs of developers or applications.

2 Approach Overview

Let us consider the following example derived[2] from [6] where a user wants to find all sections whose title refers to "XML", in the document below.

[1] To be presented at the ACM SIGIR'2002 Workshop on XML and IR, Tampere, Finland, Aug 2002.
[2] The original "Q9 example" in [6] consists of "finding all section or chapter titles that contain the word XML, regardless of the level of nesting" Our task here is slightly different as we are not looking for occurrences of

```
<chapter>
  <title>XML Data Model</title>
  <section>
    <title>Syntax For Data Model</title>
  </section>
</chapter>
<chapter>
  <section>
    <title>XML</title>
       <section>
          <title>Basic Syntax</title>
       </section>
  </section>
</chapter>
<chapter>
  <section>
    <title>XML and Semistructured Data</title>
  </section>
</chapter>
```

Figure 1: XML Document Example

In XQuery s/he would express a query of the form:

```
<results>
  {
    for $t in document("books.xml")//section/title
    where contains($t/text(), "XML")
    return $t
  }
</results>
```

Figure 2: XQuery Example

Even if the XQuery syntax could be made less "cryptic", the user would still need to know about elements like *section* and *title*. Worse, assume that there is no section but a chapter with a relevant title, the user would need to reformulate his query, replacing the string *//section/title* with *//chapter/title*. It would be possible to specify ahead of time that both elements qualify[3] by using the "union" operator. However, both *chapter* and *section* matches would be considered equally, even if one might be interested in chapters only if no relevant sections are found. More generally, the user would need to specify exhaustively the name of all alternate elements that would be "close enough". Note that it might be possible to automate this process by using the extension mechanism in XPath, but this would require ad-hoc programming, while we would much prefer here a generic process.

We suggest here a more user-friendly approach where users can express queries from the simplest possible way (free-text) to more complex structural constraints depending on their knowledge of the DTD. They should get not only perfect matches but also "close enough" ones ranked according to a certain measure of relevance. One key contribution of this work is to avoid defining yet another complex XML query language but rather to allow users to express their needs as fragments of XML

---

the element title, but for document units (in our case sections), hence we changed Q9 as well as the corresponding searched data.

[3] This is actually the original Q9 example in [6] where the "for" statement of our example reads: "for $t in document("books.xml")//chapter/title union document("books.xml")//section/title"

documents, or XML fragments for short. Users should not need to reformulate their queries in case they become too specific. The ranking mechanism should be responsible for giving priority to the closest form. This approach of using a very simple "fragment-based" language rather than XQuery is somehow analogous to using free-text rather than Boolean queries in IR: less control is given to the user, and most of the logic is put in the ranking mechanism so as to best match the user's needs.

We explain below how we propose to extend the vector space model so as to be able to compare XML fragments and XML documents as objects of the same nature. We then explain how we have embodied our model in a regular IR system, and briefly describe the basic principles our ranking mechanism that is key to our entire approach. Note that we do not provide a real formal evaluation yet, as evaluation experiments will be conducted in the context of INEX (Initiative for the evaluation of XML Retrieval) effort [7] later this year.

3 Extending the vector space model

Let us remind here that in the regular vector space model, documents and queries are indexed in a similar manner, so as to produce vectors in a space whose dimensions represent each a distinct indexing unit[4] t_i. The coordinate of a given document D on dimension t_i, is noted $w_D(t_i)$ and stands for the "weight" of t_i in document D within a given collection. It is typically computed using a score of the *tf x idf* family that takes into account both document and collection statistics. The relevance of the document D to the query Q, noted below $\rho(Q,D)$, is then usually evaluated by using a measure of similarity between vectors such as the cosine measure, where:

$$\rho(Q,D) = \frac{\sum_{ti \in Q \cap D} w_Q(t_i) * w_D(t_i)}{\|Q\| * \|D\|} \quad (1)$$

We propose here to use as indexing units not single terms[5] but pairs of the form (t_i,c_i), where terms are qualified by the context in which they appear. In order to identify this context of appearance, we borrow from the XPath model of XML documents – where each document is represented by a tree of nodes – its use of a path notation for navigating through the hierarchical structure of the document.

Thus, in Figure 1, the first occurrence of "XML" will be associated with the path "/Chapter/Title" as its context. We suggest then changing the similarity measure accordingly. Thus, in (1) the weight of individual terms should be replaced by a weight in context that we note $w_D(t_i,c_i)$. In addition, we propose to relax the scalar product model by accounting not only for exact "term in context" matching (orthogonal dimensions) but also for context resemblance. In other words, we suggest to increase the relevance score not only when a same (t_i,c_i) is found in the query and the document, but also when a same t_i appears in different but somehow related contexts c_i and c_j. Thus, if we note *cr* (context resemblance), the measure of resemblance between contexts, we propose to use as measure of similarity between XML fragments and XML documents:

[4] For the sake of the simplicity, we assume here that the basic indexing unit is a single term, but the approach applies also in the case of multi-word or phrase units.

[5] A desired property of the weights is that contexts can be unified so that w(ti, cj+ck) = w(ti, cj) + w(ti, ck)

$$\rho(Q,D) = \frac{\sum_{(ti,ci)\in Q}\sum_{(ti,ck)\in D} w_Q(t_i,c_i) * w_D(t_i,c_k) * cr(c_i,c_k)}{\|Q\| * \|D\|} \quad (2)$$

We impose that *cr* values range between 0 and 1, where 1 is achieved only for a pair of perfectly identical contexts. Thus, we see that (2) is identical to (1), in the special case of free-text where there is one unique default context.

3.1 A Richer Inverted Index

We have extended an existing full-text information retrieval system, Juru [8] developed at the IBM Research Lab in Haifa, so as to store the novel indexing scheme described above. At indexing time, XML documents are parsed by an XML parser and a vector of *(t,c)* pairs is extracted to create the document profile. By storing terms with their contexts, the posting-list of term *t* that encapsulates all occurrences of t in all documents, is split into different posting lists, one posting list for each of the contexts of *t*. This splitting allows the system to efficiently handle retrieval of occurrences of a term *t* under a special context *c*.

We use a scheme first introduced in [9] for navigating XML collections and implemented in the XMLFS system that allows to store such pairs *(t,c)* in the lexicon of a regular full-text information retrieval system via only minor modifications: each pair *(t,c)* is presented to the indexer as a unique key *t#c*. At retrieval time, the system can identify the precise occurrences of the term *t* under a given context *c* in the collection, by fetching the posting list of the key *t#c*. The system is also able to retrieve any subset of the contexts under which term *t* appears by merging the relevant posting lists. Juru stores all index terms (that form the lexicon of the system) in a trie data structure and therefore all contexts under which the term *t* has been stored can easily be retrieved by suffix matching of "*t#*." See [10] for more information on the trie structure.

4 Querying via XML Fragments

At retrieval time, the query is indexed and for each query pair (t_i,c_i) the algorithm retrieves all the contexts of t_i via suffix matching of t_i# as explained above.
Thus, for a query of the form:

```
<chapter><title>XML tutorials</title></chapter>
```

we produce a profile[6] of the form:

```
{(xml, chapter/title), (tutorial,chapter/title)}
```

More complex queries can also be expressed simply as "bags of words" decorated by location paths. Thus assume that the user wants documents whose titles relate to XML tutorials but whose contents deal for instance with XPath and XQuery, without really knowing with which path they should be associated, s/he can express a query of the form:

```
<article><title>XML tutorials</title></article> relating to XPath XQuery
```

then a profile of the following form will be produced:

[6] Note that the free-text part goes through the regular stemming and normalization stages before being decorated with contextual information. Also location information is kept to verify that terms appearing in the same context, also appear in the same instantiation of this context.

```
{(xml,article/title),(tutorial,article/title),
  (relate,null)(XPath,null), (XQuery,null)}
```

In addition the user should be able to use all the usual operators typically used on Web search services such as phrase operators (such as on the "XML tutorials" phrase below) and +/- operator such as on the "XPath" term below.

```
<article><title>"XML tutorials"</title></article> relating to +XPath  XQuery
```

4.1 The ranking algorithm

Following the model described in 3.1, all contexts are assigned a *cr* score according to their similarity to the query term context c_i. For each similar context *c'* (a context with a positive *cr* score *i.e.* somehow similar to "chapter/title" in our example) the algorithm retrieves the posting lists of (t_i,c'), and scans the posting lists to accumulate document scores. During accumulation, each occurrence of the term *(t,c')* within a document contributes to the document score according to the weight of the term in context as well as the resemblance between *c* and *c'*.
The accumulation algorithm is presented in details in Figure 4. Note that the algorithm is somehow simplified in the sense that classical performance optimization (such as the order of processing of postings) are disregarded in order to focus only on the scoring method.

The key parameters needed to adequately compute each document score when accumulating results are according to Formula (2), *w(t,c)* and *cr*. In order to define *cr*, we propose to represent contexts as strings of the XML elements forming their XPath and use techniques from the string and pattern matching community. Our proposed *cr* is discussed below.

4.2 Context resemblance measure

Let us consider a query of the form <q1><q2><q3>T</q3></q2></q1> expressed as an XML document fragment as suggested above. We express the context of the free-text part T as $Q=q_1/q_2/q_3$, or for simplifying our notation since we will now consider the path as a string of elements, let us note it $Q=q_1q_2q_3$. Similarly, let us note the context of occurrence of T in an arbitrary XML document, $A=a_1...a_8$. Table 1 below gives an example instantiation for Q and A.

a1	a2	a3	a4	a5	a6	a7	a8
language	media	book	chapter	section	subsection	title	number
q_1		q_2				q_3	
language		book				title	

Table 1: Examples of location paths for Q and A

- Input: $q = ((t_1,c_1),,(t_2,c_2),..(t_k,c_n))$, the profile of a given query
  N : the number of documents to retrieve
- Output: the most relevant n documents for the input query

1. **Let** AC be the accumulator for query units (t,c) and their associated postings
2. **Assign** AC ← null
3. **For each** term-context pair (*t,c)* in q
   - **Retrieve** *C*, the set of all contexts of *t // this is done by fetching the sub-tree of the trie whose root is associated with the prefix t#*
   - **If** *(c != null) // if t is not a free-text query term*
   - **Then**
     - **For each** context *c'* in *C*
       - **Compute** *cr(c',c) // the context resemblance score between c and c'*
       - **Assign** context_score*(t,c')* ← *cr(c'c)*
       - **If** *context_score(t,c')>0*
         - **Retrieve** the posting list *p(t,c')* from the index
         - **Add** <*(t,c'),p(t,c')*> to AC //*the accumulator stores the term associated not only with the exact context specified in the query also related ones*
   - **Else** *// t is a free-text query term*
     - **For each** context *c'* in *C*
       Retrieve the posting list *p(t,c')* from the index.
     - **Merge** all posting lists into one posting list, *p(t,null).*
     - **Assign** *context_score(t,null)* ← *1*
     - **Add** <(t,null),p(t,null)> to AC
4. **For each** *doc* in collection *D // initialize the score of each document to zero*
   - *Score(doc) = 0*
5. **For each** pair <*(t,c),p(t,c)*> in AC
   - **For each** posting entry *(d, OccNo((t,c),d))* in *p(t,c)*
     - **Compute** *W((t,c),d) = log(OccNo((t,c),d)+1) * log(N/Nt),* where
       N is the –number of documents in collection,
       Nt – number of documents containing (t,c) //regular free-text score
     - *Score(d) ← Score(d) +context_score(t,c) * W((t,c),d) //update the score of d*
6. **For each** *doc* in collection *D,*
   Score(doc) ← Score(doc) / |doc| // Normalize document scores by document length |d|
7. **Sort and return** the *n* documents with the highest scores.

Figure 3: Ranking algorithm for flexible queries

We suggest now to define the context resemblance between Q and A, *cr(Q,A)*, where the value of *cr* ranges between 0 and 1, with 1 corresponding to Q and A being identical. Additional desired properties of *cr* are that it should get a high value when fulfilling the following criteria:

1. The context *A* includes many of the q_i's of Q in the right order. In our example above this number equals 3 in for $a_1a_3a_7$
2. The occurrences of the q_i 's are closer to the beginning of *A* than to the tail, based on the intuition that the higher levels are more discriminatory than the lower levels in the XML document tree. In our example a match of $q_1q_2q_3$ with $a_1a_3a_7$ will be preferable to a match with $a_1a_3a_8$
3. The occurrences of the q_i's in *A* are close to each other: In our example a match of $q_1q_2q_3$ with $a_2a_3a_4$ will be preferable to a match with $a_1a_3a_5$
4. The length of A is shorter for the same number of matched elements, in other words, between two contexts which match on exactly the same q_i, the shorter one will get a higher score.

We propose to use 4 scores to account for each of these criteria, respectively:

1. <u>*LCS(Q,A):*</u> We use a classical dynamic programming algorithm in order to compute the Longest Common Subsequence (LCS) [11], between *Q* and *A* as strings. We note the length of this longest subsequence *lcs(Q,A),* and propose to use this value normalized by |Q, the length of the string Q, in order to account for criterion #1: ***LCS(Q,A) = lcs(Q,A)/|Q|.*** We easily verify that $0 \leq LCS(Q,A) \leq 1$.
2. <u>*POS(Q,A):*</u> We first compute, according to *lcs(Q,A)* what would be the average positioning of the optimal matching of *Q* within *A (*that we note *AverOptimalPosition*). This is achieved when the match starts on the first element of A and continues without gaps. Thus, in our example of Table 1, *AverOptimalPosition* = 2. Then we evaluate, using the LCS algorithm, the actual average positioning (*AP*). Note that in this case, one is looking for the leftmost alignment of the match that achieves *lcs(Q,A)*. *AP* takes the value 3.66 in our example. Last, we compute how far the actual positioning is from the optimal one, by subtracting *AverOptimalPosition* from *AveragePosition,* and normalize this difference by its possible range. Criterion 2 is thus reflected by the following factor,
   POS(Q,A) = 1-((AP-AverOptimalPosition)/(|A|-2*AverOptimalPosition+1)).
   Its value ranges between 0 and 1, with a value of 0 in the case of no match and, and a value of 1 for a leftmost match.
3. <u>*GAPS(Q,A):*</u> We propose to use another version of the LCS algorithm in order to capture the LCS alignment with minimum gaps between the occurrences of the q_i's. The interested reader should consult [12] for the details on how to compute this score that we note here *gaps.* In our previous Table 1 example, *gaps* will take a value of 4. In addition we propose to normalize it by the length of the common subsequence added to the gaps value so as to ensure that the total score will be inferior to 1. Thus, in order to account for criterion #3, we suggest a normalized value of *gaps* that we note *GAPS* and compute as: ***GAPS(Q,A) = gaps/(gaps + lcs(Q,A)).***
   This factor will take a value of zero in the case of a perfect match, as the value of *gaps* is then 0.
4. <u>*LD(Q,A):*</u> Finally, in order to give higher values to *A*'s whose lengths is similar to *Q*, we suggest to compute the length difference *ld* between A and *lcs(Q,A)* normalized by the length of A. Thus the final factor that evaluates the length difference can be

computed as: ***LD(Q,A)= (|A|- lcs(Q,A))/|A|.***
Again this factor takes a value between 0 and 1 and is 0 in the case of a perfect match between A and Q.

These scores are then combined as follows to compute the *cr* score:

$$cr(Q,A) = \alpha LCS(Q,A) + \beta POS(Q,A) - \gamma\, GAPS(Q,A) - \delta LD(Q,A)$$

Where α,β,γ and δ are positive parameters ranging between 0 and 1 that represent the comparative importance of each factor. They can be tuned but must satisfy $\alpha + \beta = 1$, so that *cr(Q,A) =1* in case of a perfect match.

The following Tables provide some examples of the effects of each factor on the total *cr* value. We take as $\boldsymbol{Q = q_1q_2q_3}$ **= book/chapter/title,** and see how various instantiations of *A* compare to *Q*. Our current experiments show that setting the parameters α,β,γ and δ to respectively 0.75, 0.25, 0.25, 0.2 gives good results[7], and thus were used in our examples below. In addition, for the sake of the readability, we isolate for each factor, its non-normalized key component, namely *lcs* (the least common substring) in *LCS*, *AP* (the average position) in *POS*, *gaps* in *GAPS* and *ld* (the length difference) in *LD.* This should allow the reader to easily verify their value.

A	lcs	AP	gaps	ld	cr
media/**book/chapter/title/**number	3	3	0	2	0.84
media/**chapter/book/title/**number	2	3	0	3	0.53
media/**title/chapter/book**/number	1	2	0	4	0.29
magazine/volume/article/**title**/number	1	4	0	4	0.19

Table 2: Effect of the least common substring (*lcs*) on *cr*

A	lcs	AP	gaps	ld	cr
book/chapter/title/subtitle/number	3	2	0	2	0.92
media/**book/chapter/title/** number	3	3	0	2	0.84
media/catalog/**book/chapter/title**	3	4	0	2	0.75

Table 3: Effect of the average position (*AP*) on *cr*

A	lcs	AP	gaps	ld	cr
media/catalog/**book/chapter/title/**subtitle/number	3	4	0	4	0.78
catalog/**book**/chapters/**chapter**/section/**title**/number	3	4	2	4	0.68

Table 4: Effect of the gaps (*gaps*) on *cr*

[7] Note though that more extensive experimentation is needed to decide on the ideal parameters.

A	lcs	AP	gaps	ld	cr
book/chapter/title/subtitle/subtitle/ number/bullet	3	2	0	4	0.88
book/chapter/title/subtitle	3	2	0	1	0.95

Table 5: Effects of the length difference (*ld*) on *cr*

In the three previous examples above, the value of *lcs(Q,A)* was equal to the length of *Q* yielding high grade for the corresponding contexts. If we replace in the various instantiations of *A* in Tables 3,4,5, the element "chapter" with the element "section" so as to change the value of lcs(Q,A) to 2, we immediately see a clear decrease in the *cr* score (for the same "**book/chapter/title**" query) as shown in Table 6,7,8.

A	lcs	AP	gaps	ld	cr
book/section/**title/**subtitle/number	2	2	1	3	0.51
media/**book/**section/**title/**number	2	3	1	3	0.45
media/catalog, **book/s**ection/**title**	2	4	1	3	0.39

Table 6: Effects of *AP* with a smaller *lcs*

A	lcs	AP	gaps	ld	cr
book/**book/**section/**title/**subtitle	2	4	1	3	0.45
book/chapters/section/subsection/**title**	2	4	3	3	0.38

Table 7: Effects of *gaps* with a smaller *lcs*

A	lcs	AP	gaps	ld	cr
book/section/**title**	2	2	1	1	0.54
book/section/**title/**number/letter/bullet	2	2	1	4	0.5

Table 8: Effects of *ld* with a smaller *lcs*

5 Some examples of XML fragments queries derived from INEX

We have implemented this support for XML fragments querying into our Juru search engine. We show in the next two figures some examples of results for two queries derived from the INEX collection. In the first example (Figure 4), the user specifies that the phrase "description logics" should appear in the abstract, but does not indicate any specific context for the other query terms "ABox", "TBox", and "reasoning". The following example in Figure 5 shows a slightly more complex query where the full query is encapsulated by the "article" element.

For both examples, ranked results are returned in the same way they would be in a regular full-text IR system. One small difference, from the GUI viewpoint, is that the user can pick here, at query time, the elements that s/he wants to appear in the result snippets, which are in both cases, the journal name (ti), the year of publication of the article (yr) and the article title (atl).

Our preliminary experiments show nice results on simple queries as such. The next step for us is to verify via users' studies that the approximate matching on paths evaluated through our *cr* measure will improve the usability of our model as well as the quality of our rankings.

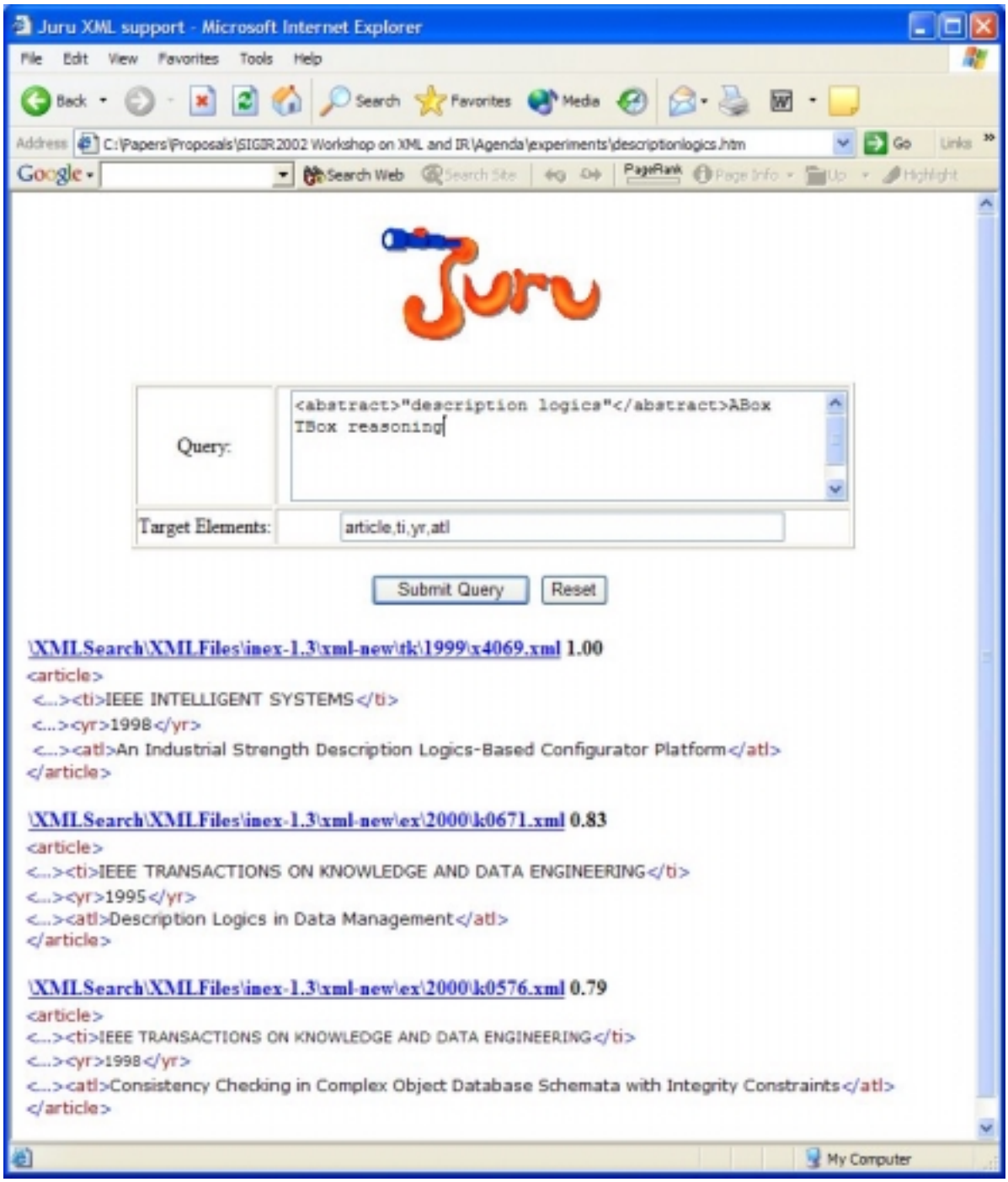

Figure 4: Example of search results for an XML fragment query derived from INEX

Figure 5: A more complex XML fragment query

6 Conclusion and Future Work

We have presented here an approach for XML search that focuses on the information needs of users and therefore addresses the search issue from an IR viewpoint. In the same spirit as the vector space model where free-text queries and documents are objects of the same nature, we suggest that query be expressed in the same form as XML documents, so as to compare "apples and apples". We have presented an extension of the vector space model that integrates a measure of similarity between XML paths, and have defined a novel ranking mechanism derived from this model. We have embodied this model in our state-of-the-art information retrieval system, Juru. While the informal experiments we have conducted are encouraging, we are now waiting for the INEX evaluation to validate our approach.

Acknowledgements

We are grateful to the XMLFS team, Benny Mandler, Naama Kraus, Alain Azagury and Michael Factor, for adding the original lexicon encoding of XML paths to Juru and for fruitful brainstorming on XML stores. Matan Mandelbrod deserves special thanks for implementing a significant part of the XML support in Juru. Finally, we thank Dafna Sheinwald for her last minute reviewing.

References

1. The XML industry portal, http://www.xml.org
2. XQuery – The XML Query language, http://www.w3.org/TR/2002/WD-xquery-20020430
3. N. Fuhr and K. GrossJohann, "XIRQL: A Query Language for Information Retrieval in XML Documents". In *Proceedings of SIGIR'2001*, New Orleans, LA, 2001
4. ACM SIGIR'2000 Workshop on XML and IR, SIGIR Forum, 2000
5. R. Baeza-Yates, D. Carmel, Y. Maarek and A. Soffer (eds), JASIST Special Issue on XML and Information Retrieval, 53: 6, 2002
6. D. Chamberlin, P. Fankhauser, M. Marchiori and J. Robie, XML Query Use Cases, W3C Working Draft 20 Dec 2001, http://www.w3.org/TR/2001/WD-xmlquery-use-cases-20011220
7. The Initiative for the Evaluation of XML retrieval, http://qmir.dcs.qmw.ac.uk/INEX/
8. D. Carmel, E. Amitay, M. Herscovici, Y. Maarek, Y. Petruschka and A. Soffer, "Juru at TREC 10 - Experiments with Index Pruning", In Proceedings of NIST TREC 10, Nov 2001.
9. A. Azagury, M. Factor, Y. Maarek and B. Mandler, "A Novel Navigation Paradigm for XML Repositories", pp 515-525 in [4]
10. A. Azagury, M. Factor, N. Kraus, I. Loy and B. Mandler, "Index Infrastructure for an XML repository", in the ACM SIGIR 2002 XML and IR workshop notes, Tampere, Finland, Aug 2002.
11. D. S. Hirschberg, "A Linear Space Algorithm for Computing Maximal Common Subsequences", *Communications of the ACM*, 18: 6, pp 341--343 (1975).
12. E. W. Myers, "Incremental Alignment Algorithms and their Applications", *Tech. Rep. 86-22, Dept. of Computer Science, U. of Arizona,* (1986).