Automatically Generating Structured Queries in XML Keyword Search

Felipe da C. Hummel[1], Altigran S. da Silva[1],
Mirella M. Moro[2], and Alberto H.F. Laender[2]

[1] Departamento de Ciência da Computação
Universidade Federal do Amazonas
Manaus, Brazil
{fch,alti}@dcc.ufam.edu.br
[2] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
{mirella,laender}@dcc.ufmg.br

Abstract. In this paper, we present a novel method for automatically deriving structured XML queries from keyword-based queries and show how it was applied to the experimental tasks proposed for the INEX 2010 data-centric track. In our method, called StruX, users specify a schema-independent unstructured keyword-based query and it automatically generates a top-k ranking of schema-aware queries based on a target XML database. Then, one of the top ranked structured queries can be selected, automatically or by a user, to be executed by an XML query engine. The generated structured queries are XPath expressions consisting of an entity path (e.g., `dblp/article`) and predicates (e.g., `/dblp/article[author="john" and title="xml"]`). We use the concept of entity, commonly adopted in the XML keyword search literature, to define suitable root nodes for the query results. Also, StruX uses IR techniques to determine in which elements a term is more likely to occur.

Keywords: XML, Keyword Search, XPath, Entities.

1 Introduction

Specifying queries through keywords is currently very common. Specially in the context of search engines on the *World Wide Web*, users with different levels of computer skills are familiar with keyword-based search. As a consequence, such a concept has been exploited outside the scope of the Web. For example, in relational databases, several methods [2, 10, 1, 8, 3, 21] have been proposed. Considering the vast number of applications that use such databases, it is clear why there is so much interest in adopting such an approach to develop more intuitive interfaces for querying in this environment.

Likewise, recently, there has been an increasing interest in the field of keyword-based search over XML documents, given the growth and consolidation of such a standard. Many techniques employ the concept of *Lowest Common Ancestor* (LCA) [7] with variations to specific requirements, including Meaningful LCA (MLCA) [17], Smallest

S. Geva et al. (Eds.): INEX 2010, LNCS 6932, pp. 194–205, 2011.

LCA (SLCA) [24], Valuable LCA (VLCA) [14], XSeek [18] and Multiway-SLCA [23]. Another structure-related concept, *Minimum Connecting Trees* [9], has led to a different approach to the problem. All of these methods aim at finding, inside the XML document, subtrees in which each query term occurs at least once in one of its leafs, and then return the root node of the subtrees as a query result. Specifically, LCA-based methods make restrictions on the choice of the root node. Notice that, for one-term queries, such methods tend to return a single-element subtree, a query result not desired in general.

Furthermore, after an initial indexing phase, they disregard the source XML document, as any query will be answered considering its own indexes only. This behavior may not be suitable in a dynamic environment in which data is frequently updated or when XML data is stored in a DBMS. Considering such an environment, it becomes relevant to develop XML keyword search methods that can easily cope with data stored and managed by a DBMS. For instance, as current XML-aware DBMS can perform XQuery and XPath queries in an efficient way, one could use that to abstract the frequent updates and storage of XML data.

This paper presents a novel method for keyword search over XML data based on a fresh perspective. Specifically, our method, called *StruX*[1], combines the intuitiveness of keyword-based queries with the expressiveness of XML query languages (such as XPath). Given a user keyword query, *StruX*is able to construct a set of XPath expressions that maps all possible interpretations of the user intention to a corresponding structured query. Furthermore, it assigns each XPath expression a score that measures its likelihood of representing the most appropriate interpretation of the user query. Then, a system can submit one or more of such queries to a DBMS and retrieve the results. Like in [21], the process of automatically transforming an unstructured keyword-based query into a ranked set of XPath queries is called *query structuring*.

This paper is organized as follows. Section 2 summarizes related work. Section 3 presents an overview of background concepts and introduces *StruX*. Section 4 describes how *StruX* was applied to the experimental tasks proposed in INEX 2010 data-centric track. Section 5 discusses the experimental results, and Section 6 concludes the paper.

2 Related Work

The basic principle behind *StruX* is similar to LABRADOR [21], which efficiently publishes relational databases on the Web by using a simple text box query interface. Other similar systems for searching over relational data include SUITS [6] and SPARKS [19]. Our proposal is different from those for two reasons: it works for a different type of data (XML instead of relational) and does not need any interaction with the user after she has specified a set of keywords. Thus, this section focuses on XML keyword search.

Using keywords to query XML databases has been extensively studied. Current work tackles keyword-based query processing by modeling XML documents as labeled trees, considers that the desired answer is a meaningful part of the XML document and retrieves nodes of the XML graph that contain the query terms. In [11] and [24] the authors suggest that trees of smaller size bear higher importance. Following such an idea,

[1] The name StruX is derived from the latin verb form *struxi*, which means to structure or build things.

Algorithm 1. *StruX* Processing

```
1:  procedure StruX(Input: unstructured query U, schema tree T)
2:      segs ← GenerateSegments(U)
3:      combs ← GenerateCombinations(segs)
4:      for each combination c in combs do
5:          cands ← GenerateCandidates(c)
6:          for each candidate d in cands do
7:              rank ← CalculateScores(d, S.root)
8:              localRank.add(rank)                    ▷ sorted add
9:          globalRank.add(localRank)                  ▷ sorted add
10:     for i from 1 to k do
11:         topK_ranks ← GenerateXPath(globalRank[i])
12:     StructuredQuery ← topK_xpaths[0]               ▷ the top query
```

XSEarch [5] adopts an intuitive concept of *meaningfully related sets of nodes* based on the relationship of the nodes with its ancestors on the graph. XSEarch also extends the pure keyword-based search approach by allowing the user to specify labels (element tags) and keyword-labels in the query. In a similar direction, XRANK [7] employs an adaptation of Google's PageRank [22] to XML documents for computing a ranking score for the answer trees. A distinct approach is XSeek [18], in which query processing relies on the notion of entities inferred from DTDs. ***StruX*** shares a similar view with XSeek in this regard, but the latter does not generate queries.

In [12], the authors propose the concept of *mapping probability*, which captures the likelihood of mapping keywords from a query to a related XML element. This mapping probability serves as weight to combine language models learned from each element. Such method does not generate structured queries, as ***StruX***does. Instead, it uses the structural knowledge to refine a retrieval model.

There are also other methods that go beyond simple keyword-based search. NaLIX [16] is a natural language system for querying XML in which the queries are adjusted by interacting with the user until they can be parsed by the system. Then, it generates an XQuery expression by using Schema-Free XQuery [17], which is an XQuery extension that provides support for unstructured or partially unstructured queries. Another approach, called EASE, considers keyword search over unstructured, semi-structured and structured data [15]. Specifically, EASE builds a schema graph with data graph indexes. It also provides a novel ranking algorithm for improving the search results. Finally, LCARANK [4] combines both SLCA and XRank for keyword-based search over XML streams. Although these approaches work on XML keyword-based queries, their goals are slightly different from our work, since they consider broader perspectives (i.e., natural language, graph-oriented data and XML streams).

3 *StruX*

This section presents ***StruX***, our method for generating structured XML queries from an unstructured keyword-based query. First, it gives an overview of ***StruX*** and then it details each of its steps.

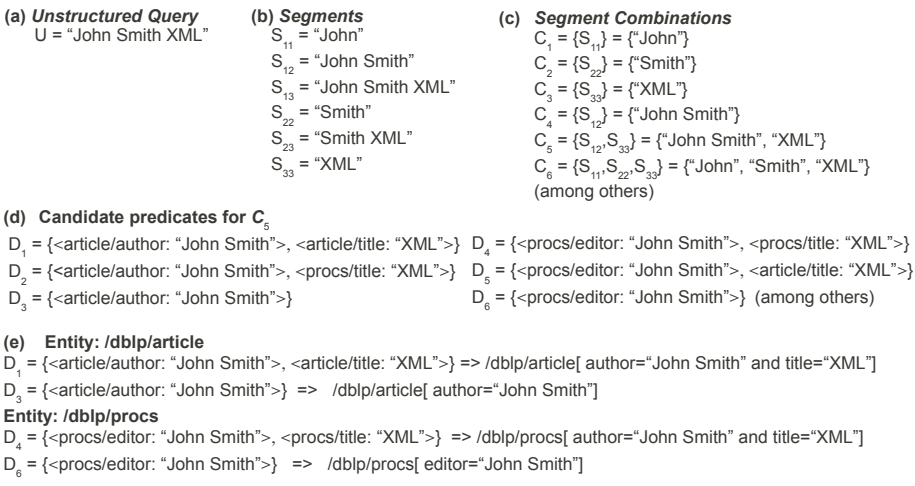


Fig. 1. Example of *StruX* steps

3.1 Overview

Algorithm 1 describes *StruX* general steps. Next, we describe each step using an example shown in Fig. 1. Given an unstructured keyword-based query as input (Fig. 1a), *StruX* first splits the input sequence of keywords into segments (Fig. 1b) and then generates combinations of these segments (Fig. 1c) to obtain possible semantic interpretations of the input unstructured query. This process assumes that each keyword-based query is an *unstructured query* U composed of a sequence of n terms, i.e., $U = \{t_1, t_2, ..., t_n\}$. This assumption is based on the intuition that the user provides keywords in a certain order. For example, a keyword query "John Clark Greg Austin" is probably intended to represent the interest in two persons named "John Clark" and "Greg Austin", respectively. But we cannot say the same for the query "John Austin Greg Clark". Although both queries have the same terms, the order in which they are specified may be used to describe a different intention. Also, this intuition helps *StruX* dealing with possible ambiguous keywords.

The next step labels segment combinations with element names from the target XML database, forming sets of element-segment pairs (Fig. 1d), or *candidate predicates*. Once these candidate predicates have been formed, *StruX* finds adequate *entities* for each candidate (Fig. 1e). In fact, *StruX* relies on the concept of *entity* [18, 20] in order to intuitively represent real-world objects. For this task, it uses a few simple heuristics.

For instance, consider an element y that has multiple sub-elements x, then y is considered an *entity*. This can be observed by looking at the document schema (or by traversing the actual document) and verifying that x can occur multiple times within an instance of element y. For example, considering the DTD specification of `author` as "`<!ELEMENT author (book*, curriculum)>`", `book` is a possible entity, while `curriculum`, according to this heuristic, is not.

In addition, we extend the concept of entity by adding another constraint that avoids very specific queries: an element x must have at least one direct descendant to be

considered an entity. For example, if `book` is defined as an element with two sub-elements like "`<!ELEMENT book (title, pages)>`", then it is an entity.

StruX identifies the elements in which the keywords are more likely to occur. It then computes scores for candidate structured queries. Such scores are needed to determine which XML query represents more accurately the user's intention. Finally, one or more top ranked structured queries are evaluated, returning the results to the users.

The final result of *StruX* is a query expressed in XPath[2], which specifies patterns of selection predicates on multiple elements that have some specified tree structure relationship. Hence, XML queries are usually formed by (tree) path expressions. Those expressions define a series of XML elements (labels) in which every two elements are related to each other (for example, through a parent-child relationship). Although other methods consider recursive schemas, we do not, since this kind of schema is not commonly found on the Web [13]. Also, notice that, in this paper, elements are always identified by their complete path to the document root, not only by their tag label.

3.2 Input Transformation

The first step executed by *StruX* (Algorithm 1, line 2) splits the input sequence of keywords into segments that represent possible interpretations of the query. A *segment* is a subsequence S_{ij} of terms from an unstructured query *U*, where S_{ij} contains the keywords from *i* to *j*. For example, considering the query *U* in Fig. 1a, the generated segments are shown in Fig. 1b. Notice that, following the intuition discussed in Section 3.1, we assume that users tend to specify related keywords in sequence. This intuition is captured by the segments. Therefore, sets of tokens that are not in sequence such as "`John`" and "`XML`" are not considered.

For each segment S_{ij}, *StruX* retrieves all elements in which *all* segment keywords appear at least once within a single leaf node. Segments that retrieve no elements are called *not viable* and are discarded from the structuring process. For example, the segment S_{23} ="`Smith XML`" would be considered not viable if the database includes no leaf having both "`Smith`" and "`XML`". In order to evaluate the likelihood of a segment S_{ij} occurring within an element n, *StruX* uses a function called *Segment-Element Affinity (SEA)*, which is defined by Equation 1:

$$SEA(n, S_{ij}) = \sum_{k=i}^{j} \text{TF-IEF}(n, t_k), \tag{1}$$

where, TF-IEF(n, t_k) measures the relevance of a keyword t_k for an element type n in the XML database. Such a function is similar to TF-IAF [21], which defines the relevance of a keyword with respect to the values of an attribute in a relational table. Nonetheless, *StruX* adapts the concept of "relational attributes" to "XML element type". This new measure is defined by Equation 2:

$$\text{TF-IEF}(n, t_k) = TF(n, t_k) \times IEF(t_k), \tag{2}$$

where each frequency is calculated by Equations 3 and 4, respectively.

[2] http://www.w3.org/TR/xpath.html

$$TF(n, t_k) = \frac{log(1 + f_{nk})}{log(1 + m)} \quad (3) \qquad IEF(t_k) = log\left(1 + \frac{e}{e_k}\right) \quad (4)$$

where f_{nk} is the number of occurrences of keyword t_k as element type n, m is the total number of distinct terms that occur in n, e gives the total number of distinct element types in the XML document, and e_k is the total number of element types in which the term k occurs.

Equation 1 evaluates every segment no matter its number of keywords. Note that a segment with two (or more) keywords is intuitively more selective than a segment with a single keyword. For example, S_{13} (from Fig. 1b) is more selective than segments S_{11}, S_{22} and S_{33}. Hence, we consider such heuristic and propose an advanced version for function *SEA* in Equation 5, called *Weighted SEA (WSEA)*, in which the number of keywords is used to favor more selective segments.

$$WSEA(n, S_{ij}) = (1 + j - i) \times \sum_{k=i}^{j} \text{TF-IEF}(n, t_k) \quad (5)$$

For representing all possible semantic interpretations of an unstructured query, *StruX* defines all possible combinations for a set of segments (Algorithm 1, line 3). Moreover, given a combination C_i, a keyword can belong to only one segment. For example, Fig. 1c illustrates some of the combinations for the segments in Fig. 1b.

For each segment combination C_i, *StruX* generates all possible sets of element-segment pairs (Algorithm 1, line 5). For example, using combination C_5 = {"John Smith", "XML"}, *StruX* obtains the sets of element-segment pairs illustrated in Fig. 1d, in which each set of pairs D_i is called a *candidate predicate*, or simply candidate. Note that $\langle procs/title \rangle$ in D_4 is different from $\langle article/title \rangle$ in D_5 as *StruX* identifies elements by their complete path to the root. At the end of the input transformation procedure, the set of candidates is able to represent every possible interpretation of the user's unstructured query. It is now necessary to determine which interpretation is more suitable to represent the original user's intention.

3.3 Candidate Predicate Selection

Once the candidates have been defined, *StruX* needs to find adequate entities for each candidate (Algorithm 1, lines 7 and 8). This is accomplished by using the recursive function presented in Algorithm 2. This function, called $CalculateScores$, performs a postorder traversal of the schema tree (which is given as input to Algorithm 1). During the traversal, the scores are propagated in a bottom-up fashion.

The propagation constant α (Algorithm 2, line 8) determines the percentage of a child's score that is assimilated by its parent score (bottom-up propagation). As a result, Algorithm 2 produces a rank that is then added to a local rank of entities for each candidate. All local ranks are merged into a sorted global rank (Algorithm 1, line 9).

Each entry in the rank is a structured query, containing a score, an entity element (structural constraint) and a candidate D_i (value-based predicates). The score of a structured query tries to measure how well it represents the user's original intention while

Algorithm 2. CalculateScores Function

```
1: function CALCULATESCORES(d, node)   ▷ Input d: candidate, node: node from the XML database
2:     for each child h in node.children do
3:         CalculateScores(d, h)
4:     for each element-segment e in c do
5:         if e.element = node.element then
6:             node.score ← node.score + e.score
7:     for each child h in node.children do
8:         node.score ← node.score + (α * h.score)
9:     if node.score > 0 AND node.isEntity() then
10:        Rank.add(root)
```

writing her query. By doing so, is its possible to determine which interpretations of the keyword-based query are more suitable according to the underlying database.

Next, a structured query can be trivially translated to XPath. Specifically, for each top-k structured query, *StruX* generates an XPath query statement based on the corresponding entity and the candidate predicate, as illustrated in Fig. 1e.

One important final note is that we chose to transform the keyword-based queries to XPath query statements for the language simplicity. However, *StruX* may also be extended in order to consider other XML query languages, such as XQuery.

3.4 Keywords Matching Element Names

So far, we have only discussed how our method addresses matches between keywords and the content of XML elements. Indeed, in *StruX* we regard such a match as the main evidence to be considered when evaluating the relevance of a structured query. However, to handle cases in which keywords match element labels, we use a very simple strategy: we boost the likelihood of all structured queries in which this is observed by adding a constant α to its score value.

3.5 Indexing

In order to build a structured query from user-provided keywords, *StruX* relies on an index of terms. This index is defined based on the target database. Specifically, each term is associated with an inverted list containing the occurrences of this term in the elements of the database. Such an association allows the query structuring process to evaluate where a term is more likely to occur within some element. Each term occurrence in an element contains a list of leaves (each one is assigned with a *leaf id*) in which the term occurs. Hence, our method can determine if two or more keywords are likely to occur in a same leaf node.

4 Experimental Setup

Following the INEX 2010 experimental protocol, we employed *StruX* to process the tasks on the data-centric track that considered the IMDB datasets. The execution was organized in *runs*, each one consisting in processing all topics under a certain setup.

Table 1. Description of the runs used in the experiments

Run	Structured Queries used	Target Datasets	Name
1	top-5	`"movies"`	*ufam2010Run1*
2	top-10	`"movies"`	*ufam2010Run2*
3	top-5	`"movies"`, `"actors"`	*ufam2010Run3*
4	top-5	all except `"other"`	*ufam2010Run4*
5	top-5	all	*ufam2010Run5*
6	top-10	all	*ufam2010Run6*

Specifically, given a topic T, we first generated an unstructured query U^T for this topic. Next, U^T was given as input to *StruX*, producing a list of structured queries $S_1^T, S_2^T, \ldots, S_n^T$ as a result, being each S_i^T associated with a likelihood score, calculated as described in Section 3.3. Then, we executed the top-K structured queries over the IMDB datasets. We used runs with different types of target documents to assess how the redundancy and ambiguity between entities in the datasets affect *StruX* (e.g., "Steven Spielberg" may appear in many parts of a *movie* document and also on *person* documents as director, producer or actor). The complete description of the runs is presented in Table 1. In the following, we discuss details regarding the generation and the processing of the runs.

Dealing with Entities. As we have already explained, *StruX* aims at generating structured queries that return single entities found in a collection of entities. In the IMDB datasets, every document root, such as `<movie>` and `<person>`, is intuitively an entity. However, *StruX* inherently considers a root element as not suitable to be an entity. To overcome this, we extended *StruX* to consider two *virtual root nodes*: (i) `<movies>` that has all `<movie>` elements as its descendants; and (ii) `<persons>` with all `<person>` elements as descendants. With such an extension, `<movie>` and `<person>` elements can now be considered entities.

Generating Queries from Topics. For each given topic from the data-centric track, we generated a keyword-based query to be used as input for *StruX*. In order to do so, we took the `<title>` element of the topic and applied a few *transformations*. This step is fully automated and aims mostly at dealing with topics specified using natural language expressions, such as "*romance movies by Richard Gere or George Clooney*", "*Movies Klaus Kinski actor movies good rating*", "*Dogme movies*". Specifically, we applied the following transformations:

i) simple stemming of plural terms, e.g.: movies → movie, actors → actor;
ii) removal of stop-words, e.g: by, and, to, of;
iii) disregard of terms indicating advanced search operators, such as "or";
iv) removal of terms preceded by "−", indicating exclusion of terms from the answers.

Fig. 2 illustrates a complete example of the whole process, including: the `<title>` field of a topic, the corresponding keyword-based query and a path expression generated from it. This particular path expression corresponds to the top-1 structured query

```
        Topic: <title> true story drugs +addiction -dealer </title>
    KB Query: true story drug addiction
Path Expression: /movie[ overview/plot, "true story drug addiction"]
       Result: <movie>
               <title>Happy Valley (2008)</title>
               <url>...</url>
               <overview>
               ...
               <plot> ... The real-life true story, Happy Valley
               ... that have been dramatically affected by
               prescription drug abuse leading to street drug abuse
               and addiction</plot>
               ...
               </movie>
  INEX Format: 2010012 Q0 1162293 1 2.6145622730255127 ufam2010Run1
               /movie[1]
```

Fig. 2. Example of the steps involved in processing an INEX data-centric topic with *StruX*

generated by *StruX*. The result obtained from applying this path expression over the IMDB dataset is also presented in the figure in two formats: as an XML sub-tree, and using the INEX output format. Next, we detail how this result was obtained.

Processing Structured Queries. The final result for a given run is obtained by processing the top-k structured queries against the target IMDB datasets. This could be performed by using some native XML DBMS such as *eXists-db*[3]. However, for our experiments, we developed a simple XPath matcher, which is used to match a document against a structured query. By doing so, we could directly output the results in the INEX result submission format, instead of having to transform the output provided by the DBMS. Fig. 2 illustrates the result for one of the topics using both formats.

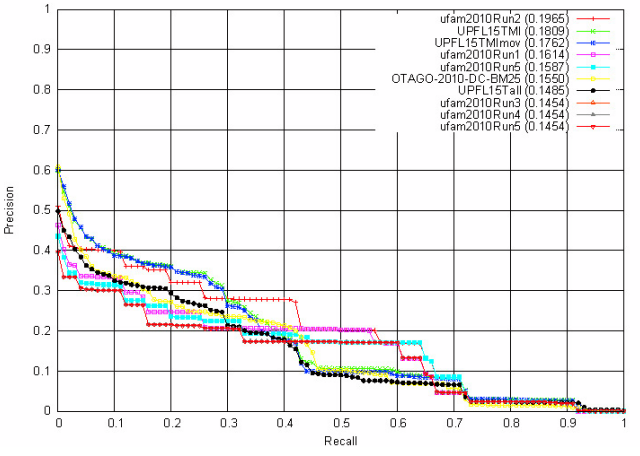
Regarding the scores of the results, as the final answers for the structured queries are produced by an external system (in our case a simple XPath matcher), there are no relevance scores directly associated to them. Thus, we simply assigned to the result the same score *StruX* has generated for the structured query from which it was obtained. Nonetheless, a single ranking of results is generated for all top-k structured queries. In this ranking, results from the top-1 query occupy the topmost positions, followed by the results from the top-2 query and so on.

5 Experimental Results

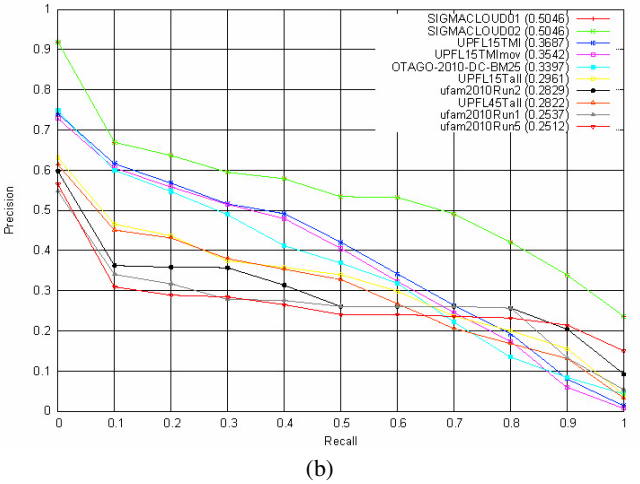
In this section, we present the results obtained in the INEX 2010 experiments.

Regarding the focused retrieval evaluation, *ufam2010Run2* was the best run among all, with MAiP value of 0.1965. As detailed in Section 4, this run returned structured queries targeting only *movies* documents.

[3] http://exist.sourceforge.net/



(a)



(b)

Fig. 3. Focused Retrieval – MAiP Metric (a) and Document Retrieval – MAP Metric (b) Results

Notice that *ufam2010Run2* uses *top-10* structured queries, while *ufam2010Run1* (fourth best run) uses only *top-5* structured queries. This illustrates that considering more structured queries does not affect the quality of the results. On the contrary, it can even improve it. This happens because many of the generated *top-10* structured queries are often not feasible, i.e., they do not retrieve any results but, on the other hand, they do not harm the final results.

Regarding the document retrieval metric, our best run was, again, *ufam2010Run2*, who achieved the seventh best MAP value among all runs. It was followed by runs *ufam2010Run1* and *ufam2010Run6*, with no significant difference between them. Runs *ufam2010Run3*, *ufam2010Run4* and *ufam2010Run5* achieved the same MAP values, meaning that adding other target document types beyond *movie* and *actor* did not affect

the overall precision. Also, resembling the focused retrieval metric, the two runs with *top-10* structured queries performed better than their counterparts with *top-5* queries.

6 Conclusions

We presented a novel method called *StruX* for keyword-based search over XML data. In summary, *StruX* combines the intuitiveness of keyword-based queries with the expressiveness of XML query languages. Given a user keyword-based query, *StruX* is able to construct a set of XPath expressions that maps all possible interpretations of the user intention to a corresponding structured query. We used *StruX* to perform the tasks proposed in the INEX 2010 data-centric track for IMDB datasets. The results demonstrated that query structuring is feasible and that our method is quite effective.

As future work, we plan to optimize even further our method. Specifically, we need to improve *StruX* for handling very large datasets. We also want to study other heuristics for improving the set of the structured queries generated. This should be accomplished by ranking the XML fragments to ensure that results closer to the original user's intention are presented first. Finally, we want to perform experiments with different Segment-Element Affinity (SEA) functions using other Information Retrieval techniques.

Acknowledgements. This work was partially supported by projects InWeb, Amanajé and MINGroup (CNPq grants no. 573871/2008-6, 47.9541/2008-6 and 575553/2008-1), and by the authors' scholarships and individual grants from CNPq, CAPES, FAPEAM and FAPEMIG.

References

1. Aditya, B., Bhalotia, G., Chakrabarti, S., Hulgeri, A., Nakhe, C., Parag, P., Sudarshan, S.: BANKS: Browsing and Keyword Searching in Relational Databases. In: Proceedings of the 28th International Conference on Very Large Data Bases, pp. 1083–1086 (2002)
2. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: A System for Keyword-Based Search over Relational Databases. In: Proceedings of the 18th International Conference on Data Engineering, pp. 5–16 (2002)
3. Balmin, A., Hristidis, V., Papakonstantinou, Y.: ObjectRank: Authority-Based Keyword Search in Databases. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, pp. 564–575 (2004)
4. Barros, E.G., Moro, M.M., Laender, A.H.F.: An Evaluation Study of Search Algorithms for XML Streams. Journal of Information and Data Management 1(3), 487–502 (2010)
5. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: A Semantic Search Engine for XML. In: Proceedings of the 29th International Conference on Very Large Data Bases, pp. 45–56 (2003)
6. Demidova, E., Zhou, X., Zenz, G., Nejdl, W.: SUITS: Faceted User Interface for Constructing Structured Queries from Keywords. In: Proceedings of the International Conference on Database Systems for Advanced Applications, pp. 772–775 (2009)
7. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: Ranked Keyword Search over XML Documents. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 16–27 (2003)

8. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient IR-style Keyword Search over Relational Databases. In: Proceedings of the 29th International Conference on Very Large Data Bases, pp. 850–861 (2003)
9. Hristidis, V., Koudas, N., Papakonstantinou, Y., Srivastava, D.: Keyword Proximity Search in XML Trees. IEEE Transactions on Knowledge and Data Engineering 18(4), 525–539 (2006)
10. Hristidis, V., Papakonstantinou, Y.: DISCOVER: Keyword Search in Relational Databases. In: Proceedings of 28th International Conference on Very Large Data Bases, pp. 670–681 (2002)
11. Hristidis, V., Papakonstantinou, Y., Balmin, A.: Keyword Proximity Search on XML Graphs. In: Proceedings of the 19th International Conference on Data Engineering, pp. 367–378 (2003)
12. Kim, J., Xue, X., Croft, W.: A probabilistic retrieval model for semistructured data. Advances in Information Retrieval, pp. 228–239 (2009)
13. Laender, A.H.F., Moro, M.M., Nascimento, C., Martins, P.: An X-ray on Web-Available XML Schemas. SIGMOD Record 38(1), 37–42 (2009)
14. Li, G., Feng, J., Wang, J., Zhou, L.: Effective Keyword Search for Valuable LCAs over XML Documents. In: Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, pp. 31–40 (2007)
15. Li, G., Feng, J., Wang, J., Zhou, L.: An Effective and Versatile Keyword Search Engine on Heterogenous Data Sources. Proceedings of the VLDB Endowment 1(2), 1452–1455 (2008)
16. Li, Y., Yang, H., Jagadish, H.V.: NaLIX: an Interactive Natural Language Interface for Querying XML. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 900–902 (2005)
17. Li, Y., Yu, C., Jagadish, H.V.: Schema-Free XQuery. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, pp. 72–83 (2004)
18. Liu, Z., Chen, Y.: Identifying Meaningful Return Information for XML Keyword Search. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 329–340 (2007)
19. Luo, Y., Wang, W., Lin, X.: SPARK: A Keyword Search Engine on Relational Databases. In: Proceedings of the 24th International Conference on Data Engineering, pp. 1552–1555 (2008)
20. Mesquita, F., Barbosa, D., Cortez, E., da Silva, A.S.: FleDEx: Flexible Data Exchange. In: Proceedings of the 9th ACM International Workshop on Web Information and Data Management, pp. 25–32 (2007)
21. Mesquita, F., da Silva, A.S., de Moura, E.S., Calado, P., Laender, A.H.F.: LABRADOR: Efficiently publishing relational databases on the web by using keyword-based query interfaces. Information Process Management 43(4), 983–1004 (2007)
22. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project (1998)
23. Sun, C., Chan, C.Y., Goenka, A.K.: Multiway SLCA-based keyword search in XML data. In: Proceedings of the 16th International Conference on World Wide Web, pp. 1043–1052 (2007)
24. Xu, Y., Papakonstantinou, Y.: Efficient Keyword Search for Smallest LCAs in XML Databases. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 527–538 (2005)