

# Index and Search XML Documents by Combining Content and Structure

Faiza Abbaci\*, Jean-Baptiste Valsamis and Pascal Francq

Université Libre de Bruxelles

Department of Information and Communication Sciences

50, Av. F. D. Roosevelt, CP 123,B-1050 Brussels, Belgium

Email : fabbaci@ulb.ac.be, jvalsami@ulb.ac.be, pfrancq@ulb.ac.be

Telephone : +32 (0)2.650.39.50

Fax : +32 (0) 2.650.39.21

**Abstract**—By nesting data, XML format allows embedding additional semantic which is not possible using flat text format. Obviously, capturing this semantic will enhance the effectiveness of the searching process in an XML corpus. Many approaches address the XML searching problem. Approaches stemmed from database communities are concentrated on the data structure. In this case, users have to express their information need in a complex query language. Moreover, users must have a good knowledge of the document structure. Approaches using information retrieval techniques are concentrated on document content. In this case the search results are not effective because the loss of the semantic conducted by the structure. This paper presents an XML retrieval system within the reach of both expert and naive users. When indexing an XML document, the system takes into account both the document content and the document structure. To query the system, a user can issue both simple queries, i.e. a bag of keywords and more complex queries using boolean operators and operators referring to document structure.

**Keywords** : XML Retrieval, Structured Search, Information Retrieval.

## I. INTRODUCTION

One of the consequences of the Internet expansion nowadays is the diversity of the data. This diversity affects both structure and content of documents. Due to the simplicity of keyword querying, users seems to be satisfied by the results returned by usual search engines.

However, web search engines index mainly plain documents as texts and HTML pages. When data is structured, or semi-structured, in opposite to plain data, the semantic is conveyed not only by the content of the data but also by its structure. 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. Nevertheless, XML query languages are very complex for a naive user and require a prior knowledge about the structure of the documents searched. Such knowledge is hardly available in Web environment. The problematic of indexing XML documents can be expressed as follows : It is well indexing the structure and the content of XML documents, in a way which preserves the document semantic and allows a simplest user query language. That is our goal when designing our system.

This paper is organized as follows. Firstly, we underline

some approaches found in the literature which attempt to resolve the problematic of XML search. Then we describe our system by giving details about the different steps performed in the search processing that is to say indexing, query evaluation and ranking.

## II. XML SEARCH APPROACHES

In the literature, XML search approaches are Database-oriented or Information Retrieval-oriented.

DB-oriented approaches are concentrating on documents structure and therefore they propose language models to querying. A language model mixes, in a syntax which is peculiar to it, conditions referring to the syntax and the content of the element (or fragment of document) searched. These approaches are inspired by SQL language. Language models are so sophisticated and complex for a naive user despite the efforts drawn up to make them more accessible. Among these approaches, we distinguish XQuery [1] which is selected as the basis for an official W3C query language for XML. We can also cite XPath [2] which is the ancestor of XQuery, Lorel query language [3] where the query is written in an SQL style, XQL [4] etc... A comparative study of some query languages is published in [5]. 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 [6] like the boolean and the probabilistic models, the vector space model is extended also to search XML documents [7]. Other approaches, uses adaptations of techniques like *tf-idf* to XML data, like for example, XSearch [8] and XRank [9] which is in a sens a generalization of the search technique used by Google [10]. In [11] the authors present XIRQL, a query language implementing IR-related concepts such as weighting and ranking. We can cite also approaches using semantic web-related concepts such as anthologies and thesauri [12], [13].

## III. AN OVERVIEW OF THE SYSTEM

When designing the system, we take into account a primordial consideration, its accessibility to the user. Thus, queries accepted by the system can vary on complexity accordingly to the user's standard. The system processes the query submitted

and returns a ranked list of fragments of documents. The resulted fragments are presented in a way that the user can navigate in the structure of the document containing these fragments. So, the user can discover the context of the information returned, this, helps him to assess the relevance of a result.

Our search engine is made up of two independent components : the first one consists of an *XML documents analyzer*. The documents go through a parser which analyzes its structure and content. The result of this analysis is stored in a database. The second component, called *query processor*, receipts the user's query, analyzes it to extract the operands when the query contains operators and finally retrieves the corresponding fragments. The architecture of our system is depicted on the figure 1.

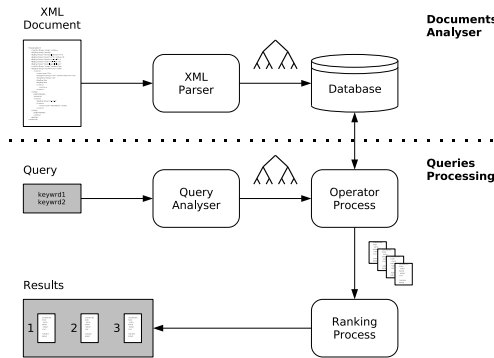


Fig. 1. Illustration of the search engine architecture

### A. Indexing XML document

In this section, we describe the details of the indexation process.

To index a document, our system performs document structure and content analysis. These operations need three index structures :

- 1) A *filemanager* table archiving the files indexed.
- 2) A *nodelist* table keeping the structure of documents archived.
- 3) A *wordlist* table keeping every word appearing in documents archived.

We choose to store the index in a MySQL database.

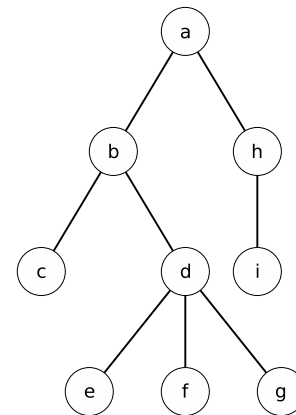
1) *Indexing document structure*: For now, we expose the simple case of a document without IDREFs. Since the storing structures of MySQL are tables that have a fixed number of columns, the idea is to transform the tree into a table, while allowing a navigation in both directions. Each node tree represents 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,
- *nbchild* 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,
- *bytepos* is the byte position in the file of the node content (the tag content for the elements and the predicate value for the attribute),
- *bytelen* is the byte length of the node content.

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 fixed in a breadth-first way, so, every children that are "brothers" are 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 figure 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 *nbchild* equal to 0. Then, the value of *child1* is :
  - 1 if the node is an element.
  - 0 for nodes representing attributes.



(a) Tree structure

ID	node	parent	nbchild	child1
0	a	0	2	1
1	b	1	2	3
2	h	1	1	5
3	c	1	0	/
4	d	1	3	6
5	i	2	0	/
6	e	4	0	/
7	f	4	0	/
8	g	4	0	/

(b) Table structure

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

2) *Indexing document content*: All words (the tag names, the attributes names, the tag contents and the attribute values) appearing in a document to index are extracted. An algorithm determines the language of the document [14], all stopwords are then removed and all remaining terms are stemmed by the Porter Algorithm [15]. These stems and their localizations are

depicted in a table named *wordlist*. A *wordlist* table contains two fields :

- *word* is the stem of a word,
- *nodes* is a string representing occurrences of the word within documents, it is in the following form :

...;#idfile<sub>i</sub>:idnode<sub>j</sub>;idnode<sub>k</sub>;...

For each document where the word appears (represented by *idfile*), we store the IDs of nodes (*idnode*) which contain this word.

### B. Query Evaluation

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 retrieved.

1) *Query Analysis*: For this search engine, a query is a set of keywords separated by operators. The matching documents must contain the words matching the query scheme. The operators include the common boolean operators and contain some more specific ones. They are described below :

- ( ) The parentheses classically change the order used to resolve the expression.

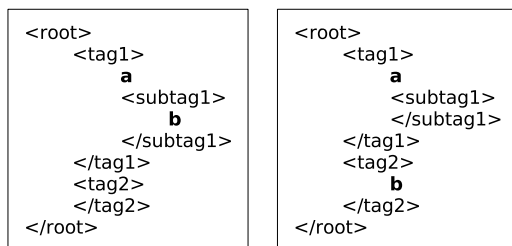
**AND** This performs the usual boolean *and* operator. It is equivalent to the '+' or the '&' symbol or when no symbol is specified.

**OR** This performs the usual boolean *or* operator. It is equivalent to the '|' symbol.

**INC** This operator is an inclusion operator, acting on the structure of the document. It corresponds to the *child* operator. An equivalent symbol is '//'. For example a INC b (figure 3(a)), matches every document that contains a in a certain element *t* and b in a sub-element of *t*.

**SIB** This is the *sibling* or the *brother* operator, also denoted by '='. a SIB b (figure 3(b)) means that a document matches if an element containing a has its parent element whose a child element (that can be the same that the one containing a) contains b.

**NOT** The negation operator. It is equivalent to '-'. This operator must always follow an operator AND, INC or SIB.



(a) Illustration of operator INC (b) Illustration of operator SIB

Fig. 3. Illustration of structure operators

Query analysis consists of parsing the queries and transforming it into a tree where nodes (excepted the leaves) are operators and leaves contain the keywords.

The query is expressed as a list of entities where each entity can be either a keyword or an operator (including opening and closing parentheses). The algorithm that parse the string is a recursive search of atomic expression whose format is *left\_expr* OP *right\_expr*. These atomic expressions are added in a tree, called *query tree*, where the operator OP is a node with two children, one by *expr*. An illustration is given on figure 4. It represents a query tree that has been constructed from the query a OR (b AND c) INC d. Note that parentheses become redundant in a tree structure since the precedence is obvious.

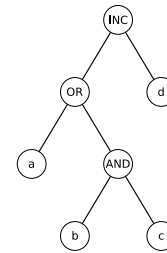


Fig. 4. Query tree associated with the query a OR (b AND c) INC d

2) *Searching the Index*: Searching the index is done in two steps. 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, the system assigns a structure, called *XQueryRes*. It is a set of *XNodeSet*, one by document in which the keyword occurs. Each *XNodeSet* contains a unique ID (called *idfile*) and a set of *XScore*. Each *XScore* 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 (see section III-C).

*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 figure 5 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 *XQueryRes* structure presented above for each query key-

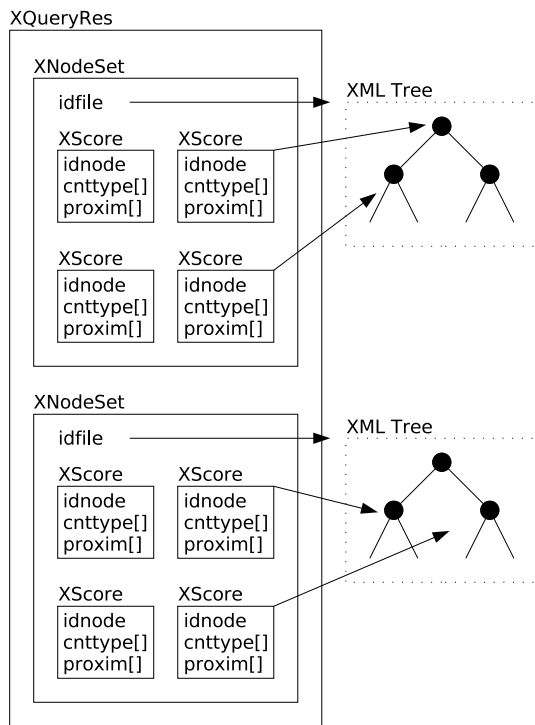


Fig. 5. Illustration of structures used to evaluate a query

word. To do so, the system creates as many XNodeSets as documents containing the keyword. And for each XNodeSet, the system creates as many XScores as nodes in the corresponding document which contains the keyword. Moreover, for each ancestor of those nodes, an XScore is created. The XScore structures of the ancestors have their value proxim incremented for each additional level.

To determine the documents (XNodeSets) matching the query, some rules are applied to merge the XQueryRes corresponding to query keywords. These rules are operations on sets of XNodeSets and sets of XScores. These operations are intersection, union, difference, ... The way to handle the ranking parameters is treated in section III-C. The intersection is used to perform an *and* operator, the union to perform an *or* and the *not* uses the difference. 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 an XScore referring to the root node. For operators which involve document structure (i.e. *INC* and *SIB*), we use an operation a bit more complex. The result of an *INC* operator is a set of XNodeSet where :

- 1) Each *idfile* appears in XQueryRes of both operands. In other words, that simply means that the keywords must appear in the same document.
- 2) These such XNodeSet must contain the *idnode* that have a *proxim* equal to 0 in the left operand and greater than 0 in the right operand.

And the result of a *SIB* operator is the set of XNodeSets

where :

- 1) As previous, each *idfile* appears in XQueryRes of both operands.
- 2) These such XNodeSet must contain the *idnode* that have a *proxim* equal to 0 in both operands.

Thus, the query tree is traversed from leaves to root and XQueryRes are merged accordingly to rules associated to the operators. When reaching the root, only one XQueryRes remains and it contains XNodeSets corresponding to the relevant documents. These XNodeSets contain in turn XScores corresponding to the relevant fragments (nodes) in the document.

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

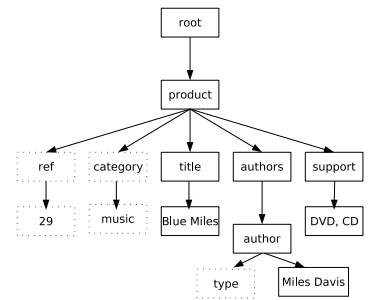
**First document *doc<sub>0</sub>* : *idfile* = 0**

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

**Second document *doc<sub>1</sub>* : *idfile* = 1**

```
<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>
```

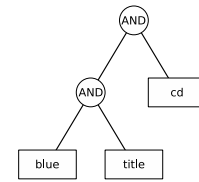
The XML-graphs associated with these documents are depicted on figure 6. Let us consider the query : *blue AND title AND cd* represented in 7(b). First, three XQueryRes are constructed for the three keywords contained in the query, namely, *blue*, *title* and *cd*. Then, the two XQueryRes corresponding to the two first keywords (*blue* and *title*) are merged into a unique XQueryRes which in turn is merged with



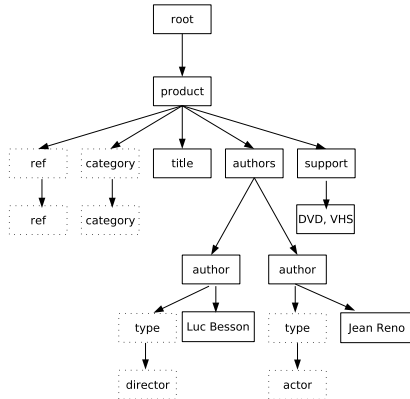
(a) XML-graph associated with the document  $doc_0$

blue AND title AND cd

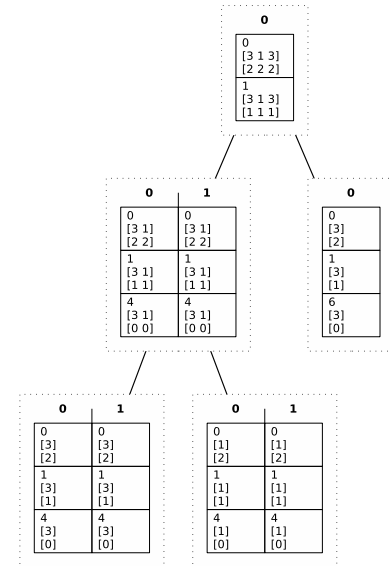
(a) Query



(b) Query tree associated



(b) XML-graph associated with the document  $doc_1$



(c) Resolution of the query tree

Fig. 6. XML-graphs

Fig. 7. An example of query evaluation

XQueryRes corresponding to the third keyword ( $cd$ ). The resulting XQueryRes (the root of the tree represented in 7(c)) contains the relevant nodes. In our example, the relevant nodes are the nodes  $0$  and  $1$  (i.e.  $root$  and  $Product$ ) of the document  $doc_0$ .

### C. Ranking results

The precedent section discusses how relevant nodes (XScores matching the query) are detected. In this section, we discuss how these nodes can be ranked accordingly to their degree of relevance to the query. In our mind, the 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 8 would be ranked higher than the fragment (b) in the same figure.

- Fourth, the hierarchical position between query keywords. The intuition related with this point is that we should take into account the semantic encapsulated in the links which relate parent and child elements. On that basis, the element (a) in figure 8 would be ranked higher than the element (b) in the same figure.
- Fifth, the relevance of the document which the element belongs. To our mind returning fragments of a document instead of the entire document is more useful to the user. Nevertheless, the relevance of a document can be used to balance the relevance of its elements. In other words, elements belonging to documents with high relevance (the relevance of a document can be the number of relevant documents it contains) should be ranked higher than elements belonging to documents with low relevance.
- Finally, The specificity of an element. The intuition here is that if a relevant element is nested in another relevant element then it (the nested element) will be ranked higher than the relevant parent element (or the relevant ancestor element).

The underlying idea is to calculate the node score using information available in the *wordlist* table, *nodelist* table, *cnttype* vector and *proxim* vector. We are currently working on determining an efficient function to calculate the score of an element (fragment) accordingly to the precedent points.

#### IV. CONCLUSION

We have presented the design and implementation of an indexing and querying method over XML documents. The distinctive feature of our indexing method is that it takes into account the keyword type (tag, content etc...) and keywords

proximity 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 determining how to combine the precedent elements in order to rank the relevant XML fragments. We hope publishing soon details of this function and an experimental study which evaluates our methodology.

#### REFERENCES

- [1] W. W. W. Consortium, "Xquery 1.0 : an xml query language," Tech. Rep., November 2003, <http://www.w3.org/TR/xquery>.
- [2] —, "Xml path language (xpath) version 1.0," Tech. Rep., November 1999, <http://www.w3.org/TR/xpath>.
- [3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, "The lrel query language for semistructured data," *JODL*, vol. 1, no. 1, pp. 68–88, April 1997.
- [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](http://www.w3.org/TandS/QL/QL98/pp/xql.html).
- [5] 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](http://citeseer.ist.psu.edu/article/bonifati00comparative.html)
- [6] 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.
- [7] 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.
- [8] 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>.
- [9] 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](http://citeseer.ist.psu.edu/guo03xrank.html)
- [10] 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](http://citeseer.ist.psu.edu/brin98anatomy.html)
- [11] 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](http://citeseer.ist.psu.edu/fuhr01xirql.html)
- [12] H. Meyer, I. Bruder, G. Weber, and A. Heuer, "The xircus search engine," 2003. [Online]. Available: [citeseer.ist.psu.edu/meyer03xircus.html](http://citeseer.ist.psu.edu/meyer03xircus.html)
- [13] 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.
- [14] P. Francq, "Collaborative and structured search: an integrated approach for sharing documents among users," Ph.D. dissertation, Université libre de Bruxelles, June 2003.
- [15] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

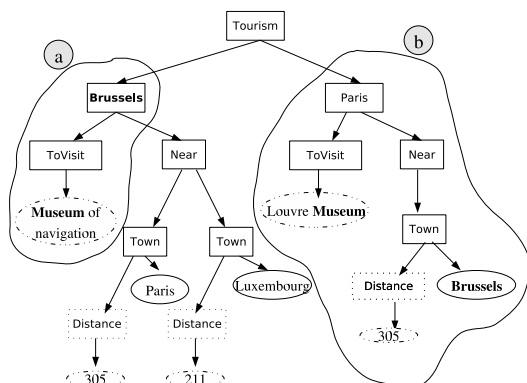


Fig. 8. Example showing keyword proximity importance