A native XML database supporting approximate match search

Giuseppe Amato and Franca Debole

ISTI - CNR
Pisa, Italy
{Giuseppe.Amato,Franca.Debole}@isti.cnr.it

Abstract. XML is becoming the standard representation format for metadata. Metadata for multimedia documents, as for instance MPEG-7, require approximate match search functionalities to be supported in addition to exact match search. As an example, consider image search performed by using MPEG-7 visual descriptors. It does not make sense to search for images that are exactly equal to a query image. Rather, images similar to a query image are more likely to be searched. We present the architecture of an XML search engine where special techniques are used to integrate approximate and exact match search functionalities.

1 INTRODUCTION

XML is becoming one of the primarily used formats for the representation of heterogeneous information in many and diverse application sectors, such as multimedia digital libraries, public administration, EDI, insurances, etc. This widespread use has posed a significant number of technical requirements to systems used for storage and content-based retrieval of XML data, and many others is posing today. In particular, retrieval of XML data based on content and structure has been widely studied and it has been solved with the definition of query languages such as XPath [1] and XQuery [2] and with the development of systems able to execute queries expressed in these languages. However, many other research issues are still open.

There are many cases where users may have a vague idea of the XML structure, either because it is unknown, or because is too complex, or because many different structures – with similar semantics – are used across the database [3]. In addition there are cases where the content of elements of XML documents cannot be exactly matched against constants expressed in a query, as for instance in case of large text context or low-level feature descriptors, as in MPEG-7 [4] visual or audio descriptors.

In the first case structure search capabilities are needed, while in the second case we need approximate content search (sometime also referred as similarity search).

In this paper we present the architecture of XMLSe a native XML search engine that allows both structure search and approximate content match to be combined with traditional exact match search operations. Our XML database can store and retrieve any valid XML document without need of specifying or defining their schema. Our system store XML documents natively and uses special indexes for efficient path expression execution, exact content match search, and approximate match search.

The paper is organized as follows. In Section 2, we set the context for our work. In Section 3 we present the overall architecture of the XMLSe system. In Section 4 we describe the query algebra at the basis of the query processor. Section 5 shows some example of query execution in terms of the query algebra. Section 6 concludes.

2 MOTIVATION AND RELATED WORK

In the Digital Libraries field three different approaches are typically used to support document retrieval by means of XML encoded metadata. The first consists in using relational database to store and to search metadata. In this case metadata should be converted into relational schemes [5] [6] [7] and this is very difficult when complex and descriptive metadata schemes such as ECHO [8] and MPEG-7 [4] should be managed: even simple XML queries are translated into complex sequences of joins among the relational tables. The second approach consists in using full text search engines [9] to index metadata records, and in general this applications are limited to relatively simple and flat metadata schemes. Besides, it is not possible to search by specifying ranges of values. The third and last approach consists in doing full sequential scan of metadata records. In this case no indexing is performed on the metadata and the custom search algorithms always scans the entire metadata set to retrieve searched information.

A relatively new promising approach is to store metadata in native XML databases as for instance Tamino [10], eXist [11], Xindice [12], which are some of software products which have been developed in recent years with this approach. However, these systems, in addition to some simple text search functionality, exclusively support exact match queries. They are not suitable to deal with multimedia metadata and to provides users with structure search functionalities.

With the continuous increase of production of multimedia documents in digital format, the problem of retrieving stored documents by content from large archives is becoming more and more difficult. A very important direction toward the support of content-based retrieval is feature based similarity access. Similarity based access means that the user specifies some characteristics of the wanted information, usually by an example image (e.g., find images similar to this given image, represents the query). The system retrieves the most relevant objects with respect to the given characteristics, i.e., the objects most similar to the query. Such approach assumes the ability to measure the distance (with some kind of metric) between the query and the data set images. Another advantage of this approach is that the returned images can be ranked by decreasing order of similarity with the query. The standardization effort carried-out by MPEG-7 [4], intending to provide a normative framework for multimedia content description, has permitted several features for images to be represented as visual descriptors to be encoded in XML.

In our system we have realized the techniques necessary to support XML represented feature similarity search. For instance, in case of an MPEG-7 visual descriptor, the system administrator can associate an approximate match search index to a specific XML element so that it can be efficiently searched by similarity. The XQuery language has been extended with new operators that deal with approximate match and ranking, in order to deal with these new search functionality.

3 SYSTEM ARCHITECTURE

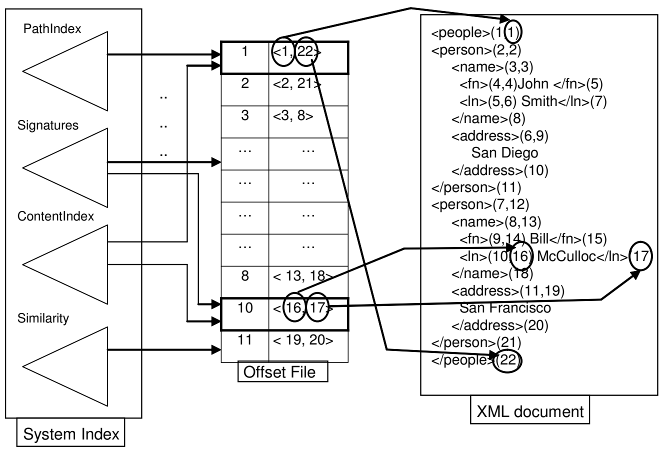
In this section we will discuss the architecture of our system, explaining the characteristics of the main components: the data storage and the system indexes. A sketch of the architecture is give in Figure 1.

3.1 Data storage

In recent years various projects [6] have proposed several strategies for storing XML data sets. Some of these have used a commercial database management system to store XML documents [5], others have stored XML documents as ASCII files in the file system, and others have also used an Object storage [13]. We have chosen to store each XML document in its native format and to use special access methods to access XML elements. XML documents are stored in a file, called *repository*. Every XML element is identified by an unique ***E**lement **I**nstance **ID**entifier* (*eiid*). As depicted in Figure 1, we use an offset file to associate every *eiid* with a 2-tuple $< start, end >$, which contain respectively a reference to the *start* and *end* position of the element in the repository. By using structural containment join techniques in [14] containment relationships among elements can be solved. The mapping between XML element names and the corresponding list of *eiid* is realized through an element index.

3.2 System Index

To improve the efficiency of XML queries, special indexes are needed. For this reason we have analyzed and realized some indexes to efficiently resolve the mapping between element and its occurrences and to process content predicates, similarity predicates, and navigation operations throughout the XML structure.



```
<people>(1,1)
<person>(2,2)
    <name>(3,3)
      <fn>(4,4)John </fn>(5)
      <ln>(5,6) Smith</ln>(7)
    </name>(8)
    <address>(6,9)
        San Diego
    </address>(10)
</person>(11)
<person>(7,12)
    <name>(8,13)
      <fn>(9,14) Bill</fn>(15)
      <ln>(10,16) McCulloc</ln>(17)
    </name>(18)
    <address>(11,19)
      San Francisco
    </address>(20)
</person>(21)
</people>(22)
```

Fig. 1. The components of data storage.

Path Index. Without special index, processing a path expression (es: *//person/ln*), with optional wildcard, involves two steps: first, the occurrences of elements specified in the path expression (es: *person* and *ln*) should be found and second, hierarchical relationships, according to the path expression being processed, should be verified with containment joins. Using ad hoc indexes, like in [15] [16] [17], which associate entire pathnames with the list of their occurrences in XML documents, processing a path expression is much more efficient.

In accord to this approach we have proposed a new path index to resolve efficiently the path expressions. The advantage of our approach with respect to the others, is that also path expressions containing wildcards in arbitrary position can be efficiently processed. This approach, discussed in [18], is based on the construction of a *rotated path lexicon*, consisting of all possible rotations of all element names in a path. It is inspired by approaches used in text retrieval systems to processing partially specified query terms. In our system the concept of term is substituted by path: each path is associated with the list of its occurrences and for this reason we call *path lexicon* the set of occurring paths (see Figure 2). Let *path*, $path_1$ be pure path expressions, that is path expressions containing just a sequence of element (and attribute) names, with no wildcards, and predicates. We can process with a single index access the following types of path expressions: *path*, *//path*, *path//*$path_1$, *path//*, *and //path//*. For more details on technique see [18].

Content Index. Processing the queries that, in addition to structural relationships, contains the content predicates (es: */people/person//ln='McCulloc'*), can be inefficient. In order to solve this problem we have extended our path index technique to handle simultaneously the content predicates and structural relationships. The content of an element is seen as a special child of an element so it is included as the last element of a path. Of course, it does not make sense to index content of all elements and attributes. The database administrator can decide, tacking into account performance issues, which elements and attributes should have their content indexed. By using this extension, an

Path lexicon	Posting List
/people	{1}
/people/person	{2,10}
/people/person/name	{3,11}
/people/person/name/fn	{4, 12}
/people/person/name/ln	{6, 14}
/people/person/address	{8,16}

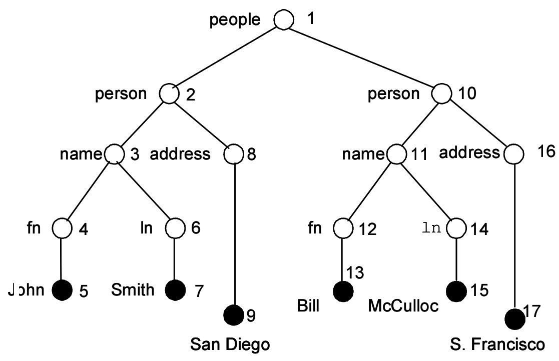


Fig. 2. The paths and their inverted lists associated

expression of comparison can simply be processed by a single access to the path index [18].

Tree Signature. Efficient processing of path expressions in XQuery queries requires the efficient execution of navigation operations on trees (ancestor, descendant, parent etc ...), for this reason in our system we have used the *s*ignature file approach. Signatures are a compact representations of larger structures, which allow the execution of queries on the signatures instead of the documents. We define the tree signature [19] as sequences of tree-node entries to obtain a compact representation of the tree structures. To transform ordered trees into sequences we apply the *preorder* and the *postorder* numbering schema. The *preorder* and *postorder* sequences are ordered lists of all nodes of a given tree T. In a *preorder* sequence a tree node is traversed and assigned its rank before its children are assigned their rank and traversed from left to right, whereas in the *postorder* sequence a tree node is traversed and assigned its rank after its children are assigned their rank and traversed from left to right.

The general structure of tree signature for a document tree T is

$$sig(T) =< t_1, post_1, ff_1, fa_1; t_2, post_2, ff_2, fa_2; \ldots; t_n, post_n, ff_n, fa_n >$$

where $ff_i(fa_i)$ is the preorder value of the first following (first ancestor) node of the node with the preorder number i. The signature of an XML file is maintained in a corresponding signature file consisting of a list of records. Through this tree signature the most significant axes of XPath can be efficiently evaluated, resolving any navigation operation.

Exploiting the capability of the tree signature is it is also possible to process *structure search queries*, as discussed in [3]. In fact, there are many cases where the user may have a vague idea of the XML structure, either because it is unknown, or because it is too complex. In these cases, what the user may need to search for are the relationships that exist among the specified components. For instance, in an XML encoded bibliography dataset, one may want to search for relationships between two specific persons to discover whether they were co-authors, editors, editor and co-author.

Approximate Match Index. Recently published papers [20] [21] investigate the possibility to interrogate XML documents not only with the exact-match paradigm but also with the approximate match paradigm. An exact-match approach is restrictive, since it limits the set of relevant and correlated results of queries. With the continuous increase of multimedia document encoded in XML, this problem is even more relevant. In fact rarely a user express exact requests on the features of a multimedia object (e.g., color histogram). Rather, the user will more likely express queries like *"Find all the images similar to this"*.

For supporting the approximate match search in our system, we have introduced a new operator $\sim$, which can be applied to content of XML elements. To be able to resolve this type of query we have used suitable index structures. With regard to the generic similarity queries the index structure which we use is the AM-tree [22]. It can be used when a distance function is available to measure the (dis)-similarity among content representations. For instance it can be used to search by similarity MPEG-7 visual descriptors.

For the text search, we make a use of the functionalities of the full-text search engine library. Specifically we have used Lucene [23].

4 QUERY ALGEBRA

An XQuery query is translated into a sequence of simple operations to be executed (the logical query execution plan). Operators of our query algebra take as arguments and return lists of tuples of *eiid* (see Section 3.1). We call these lists ***E**lement **I**nstance **I**dentifier **R**esult* (*EIIR*). For instance, given an *EIIR* R, the evaluation of R_O = *Parent(R, article)* gives back the *EIIR* R_O that is the set of elements named article, which are parents of elements contained in R.

InstanceElements. To initiate processing a query, the first step is finding the occurrences of the element names specified in the query. We define the operator $R_O = instanceElements(EN)$ that returns R_O, which contain all the *eiid* corresponding to EN, where with EN we indicate an element name. It returns all occurrences of the element name EN in the repository.

Selection. The selection $R_O = select_P(R_I)$ is applied to R_I to return $R_O \subseteq R_I$ that satisfy a selection predicate P. In addition to the standard set of operators (=, $\leq$, etc.), the elementary conditions supported by XML include the approximate match (or similarity) operator $\sim$, which is used as a binary operator as *Exp* $\sim$ *Const.* When the elements of *Exp* are indexed using the AM-tree index the selection operator returns all the elements similar *Const*, according to the similarity function associated with the AM-Tree. When the *Exp* is indexed using the full-text index, the selection returns all the elements whose content is pertinent to the text given.

Join. The join operator ($R_O = R_I \bowtie_P R_E$) take as input two *EIIR* , respectively external R_E and internal R_I, and returns the *EIIR* output R_O, which contain the elements of $R_I \times R_E$ that satisfy the predicate P, which is defined on both the *EIIR* .

Navigation operator. The navigation operators, which we evaluate using the signatures (as described in [24]), are described in the following:

- a **child** operator $R_O = child(R_I)$, which given the *EIIR* R_I returns for every element of R_I its children. For instance the node i has as the first child the node with index $i + 1$ and all the other children nodes are determined recursively until the bound ff_i is reached.
- a **parent** operator $R_O = parent(R_I)$, which given the *EIIR* R_I returns for every element of R_I its parent. The parent node is directly given by the pointer fa_i in tree signature of every element of R_I.
- a **descendant** operator $R_O = descendant(R_I)$ which given the *EIIR* R_I returns for every element of R_I its descendants. The descendants of node i are the nodes with index $i + 1$ up to nodes with index $ff_i - 1$.
- an **ancestor** operator $R_O = ancestors(R_I)$ which for every element of *EIIR* R_I returns its ancestors. The ancestors nodes is calculated like a just recursive closure of **parent**.

Particular combinations of these operation can be processed with the path index (see Section 5). For more details about realization of this operator see [19].

Structure Join. The structure join operator is used to support structure search queries. It is useful when the structure of XML data is unknown and the specific objective of the query is to verify the existence of relationships (in terms of XML hierarchies) among specific elements. Basically this operators, given a tuple of elements, verifies if they have a common ancestor below a specified level, considering level 0 that of the root of a document. For instance, in Figure 2, nodes *John* and *San Diego* have a common ancestor of level 1. On the other hand, *John* and *S. Francisco* does not have an ancestor of level 1, but they have one of level 0.

The structure join operator $R_O = structureJoin_l(R_1, \ldots, R_k)$ takes as input k *EIIR* , and returns the *EIIR* $R_O \subseteq R_1 \times \ldots \times R_k$ which have a common ancestors at least a level l in the document structure: all tuples for which there is not a common ancestor of level l are eliminated from the result.

The cost of producing first the Cartesian product of the k lists and then eliminating those tuples that do not satisfy the predicates, can be very high. In [3] we propose a new structure join algorithm, able to perform this step of query execution efficiently.

5 QUERY EXECUTION

In this section we discuss the translation of some queries XQuery into our algebra. In the following we assume that the document considered are those of Figure 2 and

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Mpeg7 xmlns="urn:mpeg:mpeg7:schema:2001" …>
<Description xsi:type="ContentEntityType">
<MultimediaContent xsi:type="ImageType">
<Image>
<MediaLocator>
<MediaUri>D:\ANSAnumb\104.JPG</MediaUri>
</MediaLocator>
<VisualDescriptor xsi:type="ScalableColorType" numOfBitplanesDiscarded="0“
numOfCoeff="64">
<Coeff>-16 34 127 94 5 14 -5 -14 27 15 -11 -28 -11 12 0 1 … </Coeff>
</VisualDescriptor>
….
<VisualDescriptor xsi:type="EdgeHistogramType">
<BinCounts>2 4 5 6 5 5 1 4 5 4 4 1 2 3 5 3 2 7 7 5 4 3 2 6 5 3 1 4 5 4 4 3 6 6 4 3 1 2 3
</BinCounts>
</VisualDescriptor>
<VisualDescriptor xsi:type="HomogeneousTextureType">
<Average>94</Average><StandardDeviation>144</StandardDeviation><Energy>238
215 186 200 189 209 210 171 179 180 179 170 174 151 122 163 123 151 144 115 98
138 128 141 139 69 53 110 61 71</Energy>
….
</Image>
</MultimediaContent>
</Description></Mpeg7>
```

Fig. 3. An example of a MPEG7 document encoded in XML.

Figure 3. We resolve the path expression with path index. We suppose that the elements *VisualDescriptor* are indexed by an AM-Tree and *ln* by a full-text index.

Example 1. Considering the following query:

```
        for $a in /people/person,
(A)     where $a//ln ~ 'Culloc'
        return $a//address
```

We look for the address of person which have *Culloc* in their lastname *ln*. This query is translated in our algebra as follow:

a) R_1 = instanceElements(*people*)
b) R_2 = child(R_1, *person*)
c) R_3 = descendant(R_2, *ln*)
d) R_4 = select(R_3, $\sim$ *'Culloc'*)
e) R_5 = ancestor(R_4, *person*)
f) R_6 = descendant(R_5, *address*)

Whereas using the indexes we have this execution plan:

A1 R_1 = PathIndex(*/people/person//ln*)
A2 R_2 = FullTextIndex(*ln*,*'Culloc'*)
A3 R_3 = Intersect(R_1, R_2)
A4 R_4 = Ancestor(R_3, *person*)
A5 R_5 = Descendant(R_4, *address*)

We have processed A1 the path expressions */people/person//ln* only with an access to index (*PathIndex*), whereas in the logical plan the same expressions is processed with three operations. Second (A2), since full-text index is available on the last element (*ln*) of path, we resolve the select operator with an access to full-text index. Then (A4) the tree signatures are used to navigate through the structure and taken first the *person* ancestor of R_3 and then the *address* descendants of R_4.

Example 2. Considering the following query related to XML document of Figure 3:

```
      for $a in /Mpeg7, $b in /Mpeg7
(B)   where $a//MediaUri ='D:\ANSAnumb\104.jpg' and
            $a//VisualDescriptor ~ $b//VisualDescriptor
      return $b
```

It returns all the elements *Mpeg7* whose visual descriptors are similar to that of image (*'D:\ANSAnumb\104.jpg'*). The logical query plan is:

a) R_1 = instanceElements$^{\$a}$(*Mpeg7*)
b) R_2 = instanceElements$^{\$b}$(*Mpeg7*)
c) R_3 = descendant(R_1, *MediaUri*)
d) R_4 = select(R_3, = *'D:\ANSAnumb\104.jpg'*)
e) R_5 = ancestor(R_4, *Mpeg7*)
f) R_6 = descendant(R_5, *VisualDescriptor*)
g) R_7 = descendant(R_2, *VisualDescriptor*)
h) R_8 = select(R_7, $\sim' R_6'$)
i) R_9 = ancestor(R_8, *Mpeg7*)

In the previous plan the instanceElements$^x(E)$ is a particular notation to indicate the recover of all the *eiid* corresponding to E, and the binding with the variable x. This is an example of a possible execution plan:

B1 R_1 = PathIndex(*/Mpeg7//MediaUri*)
B2 R_2 = Select(R_1, = *'D:\ANSAnumb\104.jpg'*)
B3 R_3 = Ancestor(R_2, *Mpeg7*)
B4 R_4 = Descendant(R_3, *VisualDescriptor*)
B5 R_5 = AM-Tree(*VisualDescriptor*, R_4)
B6 R_6 = Ancestor(R_5, *Mpeg7*)

As in Example 1 we have processed in B1 the path expressions */Mpeg7//MediaUri* with an access to index (*PathIndex*). Second (B2), we have selected from the elements of R_1, the one corresponding to the image *104.jpg*. With the navigation operations (B3, B4) we have accessed the corresponding element *VisualDescriptor*. Then (B5), since image-similarity index is available on the elements (*VisualDescriptor*), we use it to take the elements similar to the selected one R_4. Finally (B6) the tree signatures are used to navigate through the structure to access the *Mpeg7* ancestor of R_5.

6 CONCLUSION

We have presented the architecture of XMLSe, a native XML search engine that offers XML approximate content search and structure search in addition to traditional exact match search. We have introduced the various index structures that are used to efficiently process XML queries and we have presented the query algebra at the basis of the query processor. This XML search engine is particularly indicated to manage metadata for multimedia digital libraries, where approximate match queries are particularly frequent. The XML search engine has been successfully employed to support metadata management of the MILOS multimedia content management system [25], which in turns has been used for implementing multimedia digital libraries.

References

1. XPath1.0: http://www.w3.org/tr/xpath (1999)
2. XQuery1.0: http://www.w3.org/tr/xquery (2005)
3. Amato, G., Debole, F., Rabitti, F., Savino, P., Zezula, P.: Signature-based approach for efficient relationship search on xml data collections. In: XSym 2004, XML Database Symposium in Conjunction with VLDB 2004. (2004) 82–96
4. MPEG: http://www.chiariglione.org/mpeg/ (2004)
5. Florescu, D., Kossmann, D.: Storing and querying xml data using an rdbms. In: IEEE Data Engineering Bulletin Vol. 22 No 3. (1999) 27–34
6. Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., Naughton, J.: Relational databases for querying xml documents:limitations and opportunities. In: Proceedings of the 25th VLDB Conference, Edinburgh, Scotland (1999)
7. Shimura, T., Yoshikawa, M., Uemura, S.: Storage and retrieval of xml documents using object-relational databases. In: DEXA '99: Proceedings of the 10th International Conference on Database and Expert Systems Applications, Springer-Verlag (1999) 206–217

8. ECHO: http://pc-erato2.iei.pi.cnr.it/echo/ (2000)
9. Salton, G., McGill, M.: Introduction to Modern Information Retrieval. McGraw-Hill Book Company (1984)
10. Tamino: http://www1.softwareag.com/corporate/products/tamino/default.asp (2001)
11. Meier, W.: exist: An open source native xml database. In: NODe 2002 Web- and Database-Related Workshops, Springer LNCS Series 2593 (2002)
12. Xindice, A.: http://xml.apache.org/xindice/ (2001)
13. Carey, M.J., DeWitt, D.J., Franklin, M.J., Hall, N.E., McAuliffe, M.L., Naughton, J.F., Schuh, D.T., Solomon, M.H., Tan, C.K., Tsatalos, O.G., White, S.J., Zwilling, M.J.: Shoring up persistent applications. (1994) 383–394
14. Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman, G.: On supporting containment queries in relational database management systems. In: SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data, ACM Press (2001) 425–436
15. Goldman, R., Widom, J.: Dataguides: Enabling query formulation and optimization in semistructured databases. In Jarke, M., Carey, M.J., Dittrich, K.R., Lochovsky, F.H., Loucopoulos, P., Jeusfeld, M.A., eds.: VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece, Morgan Kaufmann (1997) 436–445
16. Chung, C.W., Min, J.K., Shim, K.: Apex: An adaptive path index for xml data. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002, ACM Press (2002)
17. Cooper, B., Sample, N., Franklin, M.J., Hjaltason, G.R., Shadmon, M.: A fast index for semistructured data. In Apers, P.M.G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., Snodgrass, R.T., eds.: VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy, Morgan Kaufmann (2001) 341–350
18. Amato, G., Debole, F., Zezula, P., Rabitti, F.: Yapi: Yet another path index for xml searching. In: ECDL 2003, 7th European Conference on Research and Advanced Technology for Digital Libraries. (2003)
19. Amato, G., Debole, F., Zezula, P., Rabitti, F.: Tree signatures for xml querying and navigation. In: XSym 2003, XML Database Symposium in Conjunction with VLDB 2003. (2003)
20. Fuhr, N., Großjohann, K.: XIRQL: An extension of XQL for information retrieval (2000) In ACM SIGIR Workshop On XML and Information Retrieval, Athens, Greece.
21. Guha, S., Jagadish, H.V., Koudas, N., Srivastava, D., Yu, T.: Approximate xml joins. In: SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, ACM Press (2002) 287–298
22. Amato, G., Rabitti, F., Savino, P., Zezula, P.: Region proximity in metric spaces and its use for approximate similarity search. ACM Trans. Inf. Syst. **21** (2003) 192–227
23. Lucene: http://lucene.apache.org/java/docs/index.html (2000)
24. Zezula, P., Amato, G., Debole, F., Rabitti, F.: Tree Signatures for XML Querying and Navigation. Lecture Notes in Computer Science Springer (2003)
25. Amato, G., Gennaro, C., Rabitti, F., Savino, P.: Milos: A multimedia content management system for digital library applications. In: Europeean Conference on Digital Libraries, ECDL 2004, Bath, UK, September 12-17 2004. (2004) http://milos.isti.cnr.it/.