EXTENDING AN XML MEDIATOR WITH TEXT QUERY

Clément Jamard, Georges Gardarin
Laboratoire PRiSM, Université de Versailles, 45 avenue des Etats-Unis, 78000 VERSAILLES

Keywords: XML, mediation, indexing technique.

Abstract: Supporting full-text query in an XML mediator is a difficult problem. This is because most data-sources do not provide keyword search and ranking. In this paper, we report on the integration of the main functionalities of the emerging XQuery Text standard in XLive, a full XML/XQuery mediator. Our approach is to index on keywords virtual documents in views. Selected virtual documents are on demand mapped to data source objects. Thus, the mediator selection operator is efficiently extended to support full-text search on views. Keyword search and result ranking are integrated. We rank results using a relevance formula adapted to XPath, based on number of keywords in elements and distance from the searched nodes.

1 INTRODUCTION

As XQuery becomes the standard for querying XML, new needs appear to perform full-text search in XML. A task force, Buxton and Rys (2003), is currently specifying new full-text search predicates and functions to be included in XQuery, so as to express searching on multiple keywords, ranking results on relevance, searching on suffix or prefix of terms, etc. TexQuery, Amer-Yahia (2004), can be seen as a precursor of the future language.

Some text search functionalities are very common and present in most DBMSs, such as single keyword search. Data from distributed system has to be recomposed before applying text search; important functionalities often required by applications are not possible with distributed systems. These concern ranking query results, multiple conjunctive keywords searches, and searches dealing with stemming, prefix or suffix on terms. An increasing number of XQuery-based information integration platforms are available like BEA (2004), IBM DB2 (2004), Papakonstantinou (2003) or XQuare (2005). They are mostly based on a global as view architecture and support a significant subset of XQuery. At the best of our knowledge, none of them support fully XQuery Text. However, many data integration applications are full-text oriented and requires full support.

The goal of a mediator is to federate sources around an integrated architecture fulfilling the lacks of some sources. Most data sources support single word search, some multi-keyword search, but most mediated systems have different capabilities for searching full text. For example, the XLive mediator can currently query Google as a (large) virtual XML collection through a Web service wrapper. This search engine is very powerful in multi-keyword search and in ranking results, compared to common relational databases. We also federate Xyleme, Abiteboul (2002), an XML native database system that supports efficiently multi-keyword searches. All these systems have some capabilities, but none propose the full set of XQuery text functionalities. Thus, there is a strong need to integrate uniform full-text search on all sources. Moreover the integration of the ranking systems is very difficult, as all integrated system have their own ranking scheme.

In this paper, we address the problem of extending an XML mediator for querying text-oriented sources using XQuery Text. We base our implementation on XLive, Dang-Ngoc and Gardarin (2003). The XLive system integrates and query relational or XML sources in XQuery. A large subset of XQuery is supported including FLWR expressions and nested queries. Sources are wrapped in a subset of XQuery. The query runtime is dataflow-oriented and built around an extended relational algebra for XML, known as the XAlgebra. The basic idea of this algebra is to model XML documents as tuples of paths referencing virtual DOM trees, called XTuples. The mediator evaluates query plans of XAlgebra operators on collections of XTuples and constructs XQuery results.

Jamard C. and Gardarin G. (2006).
EXTENDING AN XML MEDIATOR WITH TEXT QUERY.
In *Proceedings of WEBIST 2006 - Second International Conference on Web Information Systems and Technologies - Internet Technology / Web Interface and Applications*, pages 38-45
DOI: 10.5220/0001246000380045

Data retrieved through XLive are distributed on multiples sources. An important issue in integrating full-text search in XLive is the management of sources capabilities. We propose to unify these capabilities through views. The mediator defines views of distributed XML data and provides XQuery Text support through these views. The mediator does not materialize the views to avoid replicating sources data; but, it indexes their contents and structures. Several indexing schemes have been proposed in centralized systems for fast retrieval of elements on keywords. The interested reader can find a survey in Gardarin and Yeh (2003). We propose an efficient distributed indexing scheme that relies on a viewguide, an invariant abstract DTD-like summary derived from the query defining the view. This index scheme is particularly adapted for text search over distributed data.

XQuery/IR, Bremer and Gertz (2002), is an efficient integration of information retrieval techniques within XQuery. It uses an indexing scheme adapted to XML tree structures allowing solving tree pattern queries. Such a system does not provide a solution for our mediation context as data structures are centralized and homogeneous. Another approach to support XQuery Text query is to define function operators. TexQuery, Amer-Yahia (2004), uses Boolean operators on XML data flows to determine the presence of keywords in elements and distance between keywords. Scores functions are also defined as operators to rank results. Such operators are not easy to adapt to mediation; the mediator has to manipulate a huge amount of data through complex operations. Both systems do not provide solutions to reconcile data coming from different sources before applying text search functions. Numerous works focus on reducing index size in centralized systems (Chen and al. 2003, Chung and al. 2002, Cooper and al. 2001, Milo 1999 or Kaushik and al. 2002). In summary, although functional and efficient solutions have been studied for supporting XQuery Text, they are not easily applicable to mediation systems.

Managing view requires integration of data available in different schemas. Relationship (mappings) between schemas must be specified, to determine correspondence between elements in source schemas and elements in target schemas. A lot of work has been done on unifying source schemas under a target schema. A survey can be found in Rahm and Bernstein (2001). Defining rules mapping paths from one schema to paths of another is a simple but effective approach. Our system provides for this kind of mapping techniques to create integrated views.

This paper is organized as follows. Section 2 presents the integration of indexed views to support full-text search in XLive. Section 3 develops the query processing algorithm for querying views and ranking results. Section 4 gives some experimental results of our system. Section 5 summarizes the contributions and introduces future work.

2 INDEXING VIRTUAL VIEWS

The key question in a mediation context is how to integrate a keyword path indexing technique within the distributed architecture. In mediation systems, views are often used to focus the search on relevant data source parts. To combine the power of views with keyword search, a key decision of our design is to index virtual views of sources by keywords.

We choose to index the view content. Through the index, the mediator knows the locations of terms, which helps in answering text queries efficiently. Indexing important terms avoids replicating entire sources in the mediator. It avoids huge data transfer between sources and the mediator. The index determines relevant results, which avoids complex full text search operation on data in the mediator. A compact and fast index is the focus of our approach in order to avoid managing large data sets in the mediator.

Identifiers used in index entries are mapped to objects in sources through additional structures maintained at the mediator and source layers. We use these additional structures that determine where data composing a document in the view are located. It helps in recomposing the document efficiently.

2.1 Index Overview

We choose to index view content at creation time and to maintain index when view sources are updated. Term positions in the view are memorized in the index at mediator level. The index determines relevant elements addresses; it avoids huge data transfer between source and mediator; only relevant data are transferred. Thus, the mediator does not manipulate the whole data, through complex information retrieval operations.

Identifiers used in our index reference objects on sources using intermediate structures managed by wrappers. These structures allow view data localization, extraction, and reconstruction from sources. When an update occurs on a source, the source reports to the mediator in order to update index identifiers. Triggers or periodic polling is used to detect updates on sources. The reporting functionality depends on the source wrapper.

2.2 Location of Words in Views

To index content of the view, the position of a word has to be identified precisely. A word is located by a path and a document instance of the view. We propose our own numbering scheme to encode this position. We now detail the index structure used for determining and locating the relevant view instances

2.2.1 Numbering Scheme

We first introduce the numbering scheme implemented for identifying virtual nodes in a view. Any element in a view instance is addressed by a global document identifier (GDID) plus a node identifier (NID) determining the path reaching the element.

Global document identifier (GDID): Unique integer allocated by the mediator identifying a virtual document instance of a view.

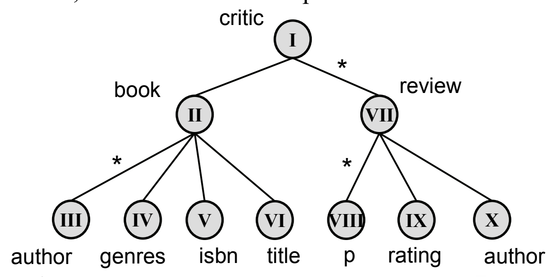
Node identifier (NID): Unique identifier of a node element in a document determined by the view definition.

To encode a NID, we make use of a *Viewguide* summarizing the structure of a view.

Viewguide: Tree giving the common structures of all documents in a view, whose nodes correspond to elements or attributes and edges to simple or multi-valued (marked with*) imbrications of elements.

Attributes are treated as elements with a name prefixed by @. All children of a node have different names as duplicates are removed. In addition, edges are marked by the maximum cardinality of the element (1 by default, and * if multiple). Thus, each distinct path of the view is represented once and only once in the viewguide.

The viewguide is somehow similar to a DataGuide (Widom and al), but: (i) It is a pure structural summary. (ii) It is derived from a view definition (i.e., the query defining the view) and not from instances. (iii) It is annotated with cardinalities of elements. It is used to assign a compact and stable unique node identifier (NID) to each element of an instance of a view. Viewguide nodes are numbered by means of a preorder traversal (see figure 1). We select this structure as it is easy to derive from a query with a fully specified return clause. View definitions are restricted to fully specified return clauses, as detailed in the sequel.



```
for $b in collection("catalog")/book
return
<critic>
  <book>
       {   for $a in $b/author
          return <author> { $a/text () } </author>
       }
       <genres> { $b/genres/text() } </genres>
       <isbn> { $b/@isbn/text () } </isbn>
       <title> { $b/title/text () } </title>
  </book>
  { for $rev in collection("review")/review
    where $b/@isbn = $rev/book/@isbn
    return
      <review>
        {   for $p in $rev/book/p
            return <p> { $p/text () } </p>
        }
        <rating> { $rev/book/rating/text () } </rating>
        <author> { $rev/book/author/text () } </author>
      </review>
  }
</critic>
```

Figure 1: View definition XQuery and ViewGuide.

To facilitate logical operation and XPath encoding, we implement a structure for NIDs. A NID is composed of a prefix and an optional suffix:

- The *prefix* is the node number assigned to the node in preorder traversal of the viewguide.
- The *suffix* corresponds to the cardinality of the traversed multi-valuated elements from the root to the node.

The document identifier determines a document instance in the view while the node identifier encodes the path to reach the node from the document root. Then a <GDID-NID> pair identifies a unique element in the view.

For example, an element identified by the XPath *critic/review/p* is assigned the path I/VII/VIII. Only the leave number is kept as node identifier, i.e., VIII. Nodes with edges marked with multiple occurrences are additionally identified by a suffix added to the identifier. Therefore the path *critic/review[1]/p[2],* which corresponds to the numbering I/VII[1]/VIII[2] is encoded as VIII[1,2]. We keep suffixes only for multi-valuated elements; a mono-valuated element has no suffix, for example *critic/book/title* is encoded as VI. Such identifiers are compact and do not change while the view definition does not change. The <GDID-NID> pair identifying the position of the *author* of the second *review* in the second document of the view is *<2-X[2]>*.

To translate XPath expressions selecting several nodes in path identifiers, we introduce the concept of identifier pattern (NID pattern called NIP). This structure is used further for query processing.

Node identifier pattern (NIP): Profile of node identifier with * in place of indices, meaning that any indice is valid.

A node identifier pattern is simply a node identifier in which stars replace one or more indice of the suffix. A star in a suffix means that any number is valid. For example, the path *critic/review/p[1]*, selecting the first *p* element in any *review* of a *critic* will be encoded VIII[*,1].

2.2.2 Word Index

The mediator stores words positions in the view in the Word Index.

Word Index: B-tree structure giving for each keyword the virtual addresses of the nodes containing these keywords.

The *Word Index* is a classical inverted list addressing element locations in virtual documents. An address is a <GDID-NID> pair determined by our numbering scheme. Keywords are determined by a thesaurus giving important words to be indexed, which can be used in queries. It is populated with location of all words at view creation time.
In a more detailed way, entries of the word index are pairs (term, position record). The position record is a table with column GDID, NID prefix, sorted list of NID suffix. Each tuple corresponds to an element containing the term with possibly multiple instances if the element is multi-valued. Table 1 illustrates two position records. This structure has been selected for fast evaluation of intersection and union operations detailed further in query processing section.

Table 1: Two position records, rec1 and rec2.

GDID	Prefix	Suffix list
120	VIII	(1,4) (3,5)
120	IX	(2)
121	VI	-
121	VIII	(3,4) (4,1)

GDID	Prefix	Suffix list
120	IV	-
120	VI	-
120	VIII	(1,2) (2,3) (2,5)
120	IX	(1)

2.3 Location on Data Sources

The Source Map maintains the mapping between a global document in the view and local documents in the sources used to compose the view instance. More precisely, we refer local documents through local document identifiers. This local identifier is associated to an extraction data operation.

Source Map: Mapping structure on the mediator mapping a GDID to a set of LDID composing the document.

Local document identifier (LDID): Number allocated by a wrapper allowing retrieving a part of a document in the source.

At view creation time, the view definition query is decomposed into atomic queries (queries referring to a single collection of XML documents). Each concerned wrapper rewrites the atomic query(ies) according to its local schema(s). Mapping between global schema (defined by atomic queries definition) and local schema can be given by a human or can be determined semi-automatically by schema matching algorithms. Mapping techniques used are not detailed here for lack of place.

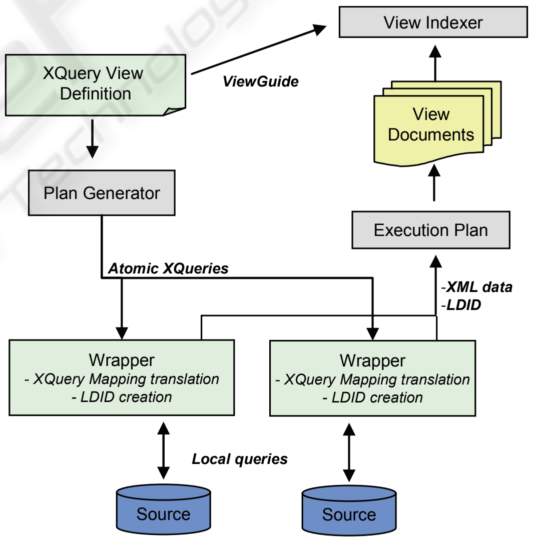


Figure 2: Framework for creating an Indexed View.

Each local source wrapper extracts and provides data to the mediator respecting the target view schema (viewguide). The view creation framework is detailed in figure 2. The *Plan Generator* defines an *Execution Plan*, which constructs view documents from data retrieved by wrappers. Data is then indexed by the *View Indexer,* which populates the Word Index and Source Map.

For each document on a local source, a LDID is created. An LDID maintains a reference to data on the local source and a reference to the mapping used to extract data. The LDID mapping depends on the wrapper. For a file wrapper, the LDID can be simply the file URI. For an XML database, it is generally a document identifier, for example a URL in Xyleme. For relational databases, it can be a reference to an SQL/XML or XQuery query allowing mapping table rows to XML. The mapping associated to each LDID determines the way to query and recompose data on local sources.

In the view example, two atomic queries corresponding to *book* and *review* are generated from the view definition. Wrappers corresponding to these collections of entities are retrieved, queries are rewritten according to local mapping, and data are extracted.

From any LDID identifier, wrappers are able to query the source to retrieve the local part of document participating in the view. Finally, the mediator uses the view definition to recompose the whole document.

3 TEXT QUERY PROCESSING

The query processing algorithm first retrieves the index entries corresponding to a textual search (e.g., search on keyword list with ranking of results). Then, it uses the retrieved node identifiers to extract from the sources the relevant elements from the view. We detail how to search the index and recompose results after querying the source. We also propose an efficient way to rank results by relevance in the context of distributed heterogeneous sources.

A multi-keyword search over semi-structured data relies on two parameters: the structural search space and a keyword list (k1, k2… kn). The structural search space defines the elements to look into. Typically, it is expressed as an Xpath. We first concentrate on conjunctive queries in which the relevant elements shall contain all keywords. Predicates in queries are of the form A1/A2…/Am[. Ftcontains k1 && k2 &&… kn], where Ai are labels (or attribute names prefixed by @) and ki are keywords. Regular expressions are allowed, i.e., Ai can be * or empty (wildcards). A typical query is “find all books in the critics view having a review dealing with XML and databases”.

Steps performed by a search are:

1. Determine the search space.
2. Compute the set of entries containing each term of the keyword list.
3. Extract documents from sources and compose results.

We define the search space using node identifier pattern(s) as define above. NIPs are derived from the Xpath expression by a viewguide traversal from the root. Notice that an Xpath query searches for all elements when the position is not specified for a repetitive element.

On the example, we rewrite the predicate Xpath in pattern *critic/review[*]*. Then, we encode the Xpath, which results in VII,(*). Thus, we obtain the root(s) of the subtree(s) interesting for the query. However, we need to access the full contents of these subtrees, to look for other keywords inside (keywords may be contained in any element among *comments*, *comment*, *p*, *author* or *rating*. To retrieve efficiently the ancestors or descendants of a node, we maintain the ancestor-descendant matrix (ad-matrix): it is a compact Boolean matrix in which element $C_{ij} = 1$ if element *i* is an ancestor of element *j* in the viewguide. This matrix provides an immediate method to retrieve the relationship between two identifiers in the viewguide.

The search space for our example query is an interval of nodes identified between VI(*) and X(*).

We query the word index to compute the set of entries containing each term of the keyword list. The strategy consists in accessing the B-tree entries (position record) for each keyword k_1 to k_n. We then intersect these lists of identifiers for conjunctive queries; the element searched must contain at least one or more occurrences of each keyword. The intersect operation determines the first common ancestor between two or more nodes. Each valid intersection (i.e., remaining in the search space) will be kept as a result.

To reduce the number of comparisons, the intersection algorithm processes intersection on entries respecting these three conditions:

1. GDID equality, i.e., keywords are in the same document. It is a trivial condition avoiding intersecting position contained in different documents.
2. NID prefixes are descendant of the NIP search space, i.e., keyword positions out of the search space are not considered.
3. NIDs suffixes are not already computed.

Figure 3 sketches the intersection algorithm. It computes every valid intersection between list of identifiers in a given search space. Condition 1 is checked on lines 8 and 12-13 to select identifiers of same documents. Condition 2 is applied when asking for next element with the search space on lines 1, 5, 13, 27. Each next element is chosen only among lines of position record corresponding to the search space. Last condition is applied on line 21-

22, by checking if the current intersection remains in the last result (descendant of that result).

Entries are ordered by NID suffix in position records, which avoids skipping intersection when scanning each list. Therefore condition on line 30 always selects the minimum NID of list to intersect.

```
   ALGORITHM INTERSECT
   INPUT           : List(n)
                          n list of keyword position
   VARIABLES :    intersect, current_id, test_id, tmp_res
                     type:identifier<GDID-NID>
                  search_space
                     type:NIP filter
                  result
                     type:list of result
1. current_id = List(0).nextElement(search_space)
2. WHILE( List(0).hasNextElement(search_space) ) DO
3.      intersect = NULL;
4.      FOR EACH LIST DO
5.            test_id = list().nextElement(search_space)
6.            /* No element intersecting in same document
7.            Repeat process on the next element */
8.            WHILE (  test_id.GDID > current_id.GDID )
9.                  current_id = test_id;
10      FOR EACH LIST DO
11           /* Find next element with same GDID */
12           WHILE (test_id.GDID < current_id.GDID) DO
13                test_id = list().nextElement(search_space);
14           /* Intersection in a given search space */
15           tmp_res =
16                  ancestor(test_id,current_id,search_space);
17.          /* Valid intersection ->
18.                 search in the next list */
19.          /* Invalid intersection ->
20.                 same process on next element */
21.          IF (tmp_res != NULL &&
22.                        !tmp_res.descendantOf(result.last))
23.               intersect = tmp_res;
24.          ELSE
25.               intersect = NULL;
26.               IF (current_id.suffix > test_id.suffix)
27.                   current_id =
28.                         list().nextElement(search_space);
29.               break;
30.
31.     /* Keep the resulting intersection */
32.     if(intersect != NULL)
33.          result.add(intersect);
34.          last_res = intersect;
35.          current_id = list().nextElement(search_space);
```

Figure 3: Intersection algorithm.

The algorithm can be applied to position record list of table 1 with the search space of our query as a NIP: VII(*). For GDID 120 only lines VIII and IX are selected. We obtain valid intersection for IX(1) records1 and VIII(1,4) records2, VIII(2,3) records1 and IX(2) records2. These are VII(1) and VII(2). Other intersections for GDID 120 are not leading to a valid intersection (in the search space, or an intersection already found). Entries of other GDID are processed in the same way.

4 XQUERY TEXT CAPABILITIES

4.1 Full Text Search

A full-text search may use the position of keywords inside the document. This position can be expressed in two metrics. The position as element means that only the path from the root of the document determines the criteria. The position inside an element means the path and the position among the other words in the element determine the criteria.

The index presented before allows answering any full-text search dealing with position expressed as element. Other functionalities like term distance, window, order or result ranking require additional information, the word offset inside an element. The offset is added to position records as needed by these operations. Full functionalities are available in Buxton and Rys (2003).

4.2 Ranking Results

A ranking method associates a relevance weight to each result. In a mediator, we have to rank results coming from different sources and to merge results for delivery to the user in correctly ranked order. Our architecture provides a way to pre-rank results, i.e., virtual results are ranked before source extraction; we compute the relevance score of each result when querying the word index. The weighting formula has to be accurate but simple enough to be computable with information contained in the word index.

We determine the weight of a result by adding the weights of each node containing directly one or more keywords. Our ranking approach is based on the specificity of each result. The ranking method gives more influence to element nodes close to the root of the search space. Thus, words close to the root weight more than words deeply hidden in the result tree. Such an approach is a bit simplistic, as ranking weights are attributed independently of keyword position in relation with each other.

We also give more influence to element nodes containing several keywords of the search. The percentage of keywords in a node is used as a polynomial factor to adjust the weight. Finally, the following formula computes the weight of a node:

$$We = \frac{Ni^{\beta}}{N} \sum_{i=0}^{n} \left(\frac{Wi}{(1+\alpha)^{d}} \right)$$

Wi is the weight of the keyword (tf.idf), *N* is the total number of keywords in the query, *Ni* is the total number of keywords in the node, and *d* is the distance to the root of the sub-graph (number of edges). The constant *α* is designed to give more influence to the distance from the root to the word position (in edges). *β* is a polynomial factor used to increase the weight of an element containing several keywords. The total weight of a result is the sum of each node weight containing one or more keywords.

An advantage of the implemented ranking system is the formula modularity. It may be extended or replaced. The mediator may integrate any formula relying on information contained in the word index (tf.idf, distance). The ranking formula is application adjustable.

Some other systems propose concrete solutions to rank results of a keyword–based query. XRANK, Lin and al. (2003), proposes a ranking algorithm relying on the *elemRank* of an XML element. This rank is computed using the number of outgoing and incoming edges (inter and intra documents). It contrasts with our approach, which focuses first on keywords distribution and second on links, by applying a tf.idf based formula on element contents. Moreover, the proximity factor proposed in XRANK is not fully adapted to XML tree structures; XRANK uses the minimum containing window (containing all keywords) as proximity metric. It is a global proximity; our system rather uses a proximity factor computed at the element granularity, i.e., not globally on the full result sub-graph. Thus, our approach is more precise on content. XRANK also uses a decreasing factor for less specific results (far from the sub-graph result root).

The XXL system, Theobald and Weikum (2002), uses relevance ranking focused on a vagueness operator, which computes a similarity score for every result. It compares the structure of the result with the structure of the query, by applying ontological rules. This kind of ranking is not easily adaptable to XQuery Text in the context of mediation, as it is not based on a specific search space. However, it could be interesting to consider ontology-based similarity for integrating heterogeneous sources.

XIRQL, Norbert and Kai (2001), extends IR functionalities to XML, like relevance-oriented search, looking for XML objects satisfying a content search. Weighting formulas are applied to objects (i.e., sub-graphs) defined in the schema at the type level and in the index at the instance level. Composed objects are weighted with the sum of the composing objects. Content queries are processed by combining relevance of objects according to the logical search conditions. Only the relevant objects are returned ranked by weights. A tf.idf computation and the specificity of the position of keywords are used to adjust the weight of the objects. This approach is difficult to apply in a mediation architecture where objects are not defined for each source.

5 EXPERIMENTS

We experiment index search on three different data sets stored in an XML repository. The data sets are presented in table 2. The size is the total size of the view after creation. Each data set is structured as the *critic* view definition given above. Collections are stored on different sources. We measure search time through the index for three queries:

(q01) critic/review [. ftcontains “k1” && … “kn”]
(q02) critic [. ftcontains “k1” && … “kn” without content .//title]
(q03) critic [. ftcontains “k1” && … “kn”]

The queries are searching documents containing a conjunctive set of keywords. The search space includes different elements of the view. q01 searches in the review element, q02 in the *review* element without *title*, and q03 in the full document. Queries are executed on the same Pentium 4 with 512K memory configuration. The numbers of keywords in queries vary from 2 to 26. The measures presented here are an average of ten executions.
Table 2 presents the execution time of q01 searching for 5 keywords. For each data set, we measure the execution time with an indexed view, and without index (the mediator handles the search operation). The time for the view includes the index search time (both Word Index and Source Map), the query plan execution in the mediator, and the result construction. The time without indexed view includes the plan execution and the result construction times. As planned, index speeds up the execution time as only relevant results are requested from the sources and complex content search operations are avoided on huge text data at the mediator layer. For each data set, the execution through the indexed view brings out a significant ratio averaging 3, for a low selectivity of queries (from 60 % to 68 % of the documents are selected).

Table 2: Data sets.

	Docs	Byte Size
ds1	100	607 855
ds2	250	1 556 052
ds3	500	3 029 990

Table 3: q01 execution time and index search time.

Execution Time		*Intersection*		
View	Mediator	5 words	15 words	25 words
1042.1	2917.6	1.6	3.1	4.6
1976.5	5969	6.3	15.6	21.9
5949	18501	17.2	65.6	86

Index search times for query Q01 are presented in table 3. The index search does not increase significantly the execution time; it represents less than 1% of the overall execution time. These preliminary results validate our approach.

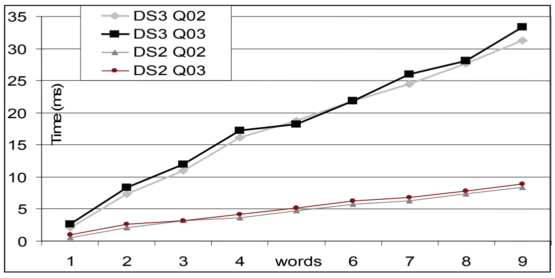


Figure 4: q02 and q03 evaluation for DS2 and DS3.

Figure 4 illustrates the index search time difference between q03 and q02. Due to the identifiers ordering scheme, q02 always executes faster than q03 as fewer elements are considered in the search space.

6 CONCLUSION

In this paper, we have reported on the integration of XQuery Text in an XML mediator. The main difficulty is to integrate sources with little capabilities in full-text search. We propose to use indexed virtual views to support such sources. The views are indexed inside the mediator using a sort of structural dataguide derived from the view definition, called a viewguide. Nodes identifiers and path expressions are encoded through the viewguide, which yields to algorithms to process efficiently the mediator basic selection operator involving XPaths and keywords. A parameterized ranking formula taking into account relevance and deepness of elements is proposed to integrate result relevance.

Further work remains to be done. Notably, a better support of source capabilities would be desirable. When a source can support a subset of XQuery, we should be able to build limited views at the wrapper to integrate it in distributed query processing. Thus, functionalities should be divided in multiple stages, e.g., concrete local views combined with global virtual views. Also, local ranking of results from a view or a capable source (e.g., Google) seems easy, but global ranking with pertinent formulas remains to be experienced in details on real applications. The propagation of updates must also be studied. Indexing structures should be automatically updated when inserting and deleting objects in data sources. A basic approach could be detecting updates at wrapper level and propagate them at the different index structures.

REFERENCES

Abiteboul S., S. Cluet, G. Ferran et M.C. Rousset: "The Xyleme project", *Computer Networks* 39(3): 225-238 (2002)

Amer-Yahia S., C. Botev, J. Shanmugasundaram : "TeXQuery: A Full-Text Search Extension to XQuery", *WWW'04*

BEA: "Liquid data for WebLogic 1.1, 2004, http://e-docs.bea.com/liquiddata/docs11/

Bremer J. M., M. Gertz : "XQuery/IR: Integrating XML Document and Data Retrieval", *WebDB 2002*.

Buxton S., Rys M. Editors, "XQuery and XPath Full-Text Requirements", W3C Working Draft 02 May 2003, http://www.w3.org/TR/xquery-full-text-requirements/

Chen Q., A. Lim and K.W. Ong : D(k)-index: An adaptive structural summary for graph-structured data. In *Proc. of SIGMOD*, 2003.

Chung Chin-Wan, J. Min and K. Shim: "APEX: an adaptive path index for XML data", *SIGMOD Conference 2002*: 121-132

Cooper B., N. Sample, M.J. Franklin, G.R. Hjaltason and M. Shadmon :" A Fast Index for Semistructured Data.", *VLDB 2001*: 341-350

Dang-Ngoc T.-T., G. Gardarin : "Federating heterogeneous data sources with XML", In *Proc. of IASTED IKS Conference*, pages 193-198, Scottsdale, USA, Nov. 2003.

Fuhr N., K. Großjohann: "XIRQL: A Query Language for Information Retrieval in XML Documents". *SIGIR* 2001: 172-180

Gardarin G., L. Yeh: "Treeguide Index: Enabling Efficient XML Query Processing", *Bases de Données Avancées*, Montpellier, Octobre 2005

IBM: "DB2 Information Integrator for Content", 2004, http://www-306.ibm.com/software/data/eip/

Kaushik R., P. Shenoy, P. Bohannon and E. Gudes : Exploiting local similarity for indexing paths in graph-structured data. In *Proc. of ICDE*, 2002.

Lin G., F. Shao, C. Botev, J. Shanmugasundaram : XRANK: Ranked Keyword Search over XML Documents. SIGMOD Conference 2003: 16-27

Milo T., D. Suciu: "Index Structures for Path Expressions", *ICDT* 1999: 277-295

Papakonstantinou Y., V. Borkar, M. Orgiyan, K. Stathatos, L. Suta, V. Vassalos, P. Velikhov : "XML queries and algebra in the Enosys integration platform", *Data Knowl. Eng*. 44(3): 299-322 (2003)

Rahm E., P.A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB journal*:334-350.

Theobald A., G. Weikum : "The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking". EDBT 2002: 477-495

Widom J. et. al.: "Lore, a DBMS for XML", http://www-db.stanford.edu/lore/

XQuare: "The XQuare project: open source information integration components based on XML and XQuery", 2004, http://xquare.objectweb.org/