Integrating Keyword Search into XML Query Processing

Daniela Florescu
INRIA Rocquencourt
France
Daniela.Florescu@inria.fr

Donald Kossmann
Univ. of Passau
Germany
kossmann@db.fmi.uni-passau.de

Ioana Manolescu
INRIA Rocquencourt
France
Ioana.Manolescu@inria.fr

Abstract

Due to the popularity of the XML data format, several query languages for XML have been proposed, specially devised to handle data whose structure is unknown, loose, or absent. While these languages are rich enough to allow for querying the content and structure of an XML document, a varying or unknown structure can make formulating queries a very difficult task. We propose an extension to XML query languages that enables *keyword search* at the granularity of XML elements, that helps novice users formulate queries, and also yields new optimization opportunities for the query processor. We present an implementation of this extension on top of a commercial RDBMS; we then discuss implementation choices and performance results.

Keywords

XML query processing, full-text index

1 Introduction

There is no doubt that XML is rapidly becoming one of the most important data formats. It is already used for scientific data (e.g., DNA sequences), in linguistics (e.g., the Treebank database at the University of Pennsylvania), to annotate large documents (e.g., Shakespeare's work), or for data exchange on the Internet (e.g., for electronic commerce). Furthermore, large software vendors, including IBM, Microsoft, and Oracle, as well as a large number of new start-ups are developing tools to manage XML data and applications which are based on XML.

One of the strengths of XML is that it can be used to represent *structured data* (i.e., records) as well as *unstructured data* (i.e., text). For example, XML can be used in a hospital to represent (structured) information about patients (e.g., name, address, birthdate) and (unstructured) observations from doctors. To take advantage of this strength, however, it is important to have tools that can work effectively with both kinds of data; it is in particular important to have XML query languages which *select* records from the structured part of an XML document and *search* for information in text. For instance, it should be possible to pose one query that finds all patients that are older than 45 years and have some specific symptoms.

Keyword search is also important to query XML data with a regular structure, if the user does not know the structure or only knows the structure partially. Such a situation arises frequently on the Web; a user visits an (XML) Web site, but does not know (and does not want to know) how the data is stored at that Web site. For instance, a user who wants to buy a car on the Internet might not know how exactly the price and category of a car are represented at the dealer's Web site; rather than looking at the DTD, the user would prefer to directly ask for all cars with *price < $1000*; this query involves a keyword search for *price* and the evaluation of a predicate on the value of *price*.

A third reason to integrate keyword search into XML query processing is to query several XML documents at the same time. Again, a user might be interested in buying a cheap car on the Internet; this time, however, the user wants to get information from several car dealers at once. The car dealers may store their data in different ways, but all car dealers that the user is interested in will somehow specify a *price* for each car. The user query will be the same as in the previous paragraph; i.e., the query will involve keyword search on *price* even if the user knows exactly how each car dealer stores his/her data.

Both regular (structured) XML query processing and keyword search have extensively been studied in previous work. (We will give an overview of related work in Section 6.) To date, however, nobody has ever shown how both features can be combined. Extending an XML query language for keyword search and showing how such an extended query language can be implemented is the purpose of this paper.

1.1 The Role of Relational Database Systems

Obviously, there are many alternative ways to process XML queries with keyword search. In this work, we propose to exploit a standard, off-the-shelf relational databases system (RDBMS) as much as possible. Examples of popular RDBMS products are IBM DB2, Microsoft SQL Server, or Oracle 8. Using an RDBMS has several advantages. First, as we will see, it is very easy to build an extended XML query processor that integrates keyword search on top of an RDBMS; it already provides most of the functionality that is required. Second, RDBMSs are universally available. Most organizations have an RDBMS installed so that no additional costs are incurred. Third, RDBMSs allow to mix XML data and other (relational) data. Not all the data in the world is XML yet! Fourth, RDBMSs show very good performance for this purpose. More than twenty years of research and development have been invested into making RDBMSs the best possible general-purpose query processors and the RDBMS vendors are continuously improving their products. In particular, RDBMSs are capable of storing and processing large volumes of data (up to terabytes).

Relational databases can be used in different ways for our purposes. In this paper, we consider two scenarios. In the first scenario the whole XML data is replicated (or initially stored) in the relational database. This scenario provides the best performance. In this scenario, the XML query including keyword search can be entirely executed by the RDBMS, thereby taking full advantage of the powerful query processing capabilities of the RDBMS and interleaving keyword search with the other operations of an XML query in the best possible way. Also, no data is moved through the network and no process boundaries need to be crossed to execute queries in this scenario. In effect, this scenario shows how an RDBMS can be used as a *data warewouse* for XML data.

Unfortunately, it is not always possible or cost-effective to build a data warehouse. In the long run, for instance, it will not be viable for technical and legal reasons to replicate all the XML data on the Web. Therefore, we describe a second scenario in which query processing is carried out in a distributed way. In this scenario, XML documents are stored by indivual data sources. An RDBMS is used to store indexes which can be used to execute keyword searches and to find all relevant XML data sources for a query. In fact, the XML data sources could again be powered by an RDBMS; however, the data sources could also be implemented on top of a simple file system.

The techniques developed in this work are also applicable if an object-oriented database system (OODBMS) is used instead of an RDBMS. To some extent, our approaches are also applicable if a special-purpose XML query processor like Tamino [Tam99] or Excelon [Exc99] is used. The current generation of OODBMSs and special-purpose XML query processors, however, is not mature enough to process large amounts of data so we focus on the use of RDBMSs throughout this paper. Furthermore, we will not exploit any "object-relational" features which are currently built into many RDBMSs because these features are not useful for our needs.

1.2 Contribution and Overview of this Paper

In a nutshell, the goal of our work is to integrate keyword search into XML query processing and make use of existing (relational) database systems as much as possible. In the remainder of this paper, we will report on the following developments:

1. We will show how to extend an existing XML query language in order to support keyword search. This will make it possible to query XML data without structure (i.e., text), help users to query XML documents with structure, if the users do not know the structure, and help to query multiple XML documents with the same ontology, but different DTDs.
2. We will present an extension of inverted files in order to support keyword search. Furthermore, we show how such extended inverted files can be stored in a relational database.
3. We will show how XML queries that involve keyword search and other operations can be entirely processed using an RDBMS, if the XML data is replicated in one relational database.
4. We will also show how XML queries with keyword search can be executed, if the XML data cannot be stored in a relational database.
5. We will present performance experiments that demonstrate the overheads of our approach (size of indexes, etc.) and give a feeling for the cost of extended XML query processing with keyword search.

Section 2 describes the data model and query language used in this work. Section 3 presents the proposed indexes for keyword search (i.e., inverted files). Section 4 discusses the role of RDBMSs in query processing in more detail. Section 5 contains performance experiments. Section 6 gives an overview of related work. Section 7 concludes this paper with suggestions for future work.

2. Data Model and Query Language

Abundant work recently addressed the problem of finding a formal data model and a query language for XML data. Since this still remains an open problem (no common agreement has been reached yet), we describe in this section the data model and query language that are the basis for our work. However, our data model and query language are similar in spirit to the other proposals so that the results presented in this paper can be easily adapted to those other formalisms. Assuming that the final standard will have the same characteristics and functionalities as the current proposals, our work will also be applicable to the future standard XML query language.
In this work we extend the XML-QL query language with keyword based search capabilities. We start this section by describing our XML data model and we continue by describing the syntax and the semantics of the current language, and of our extension.

2.1 Data Model

A relevant question is whether an XML query is evaluated on a single XML document, on a set of XML documents or on a set of XML elements. Concerning this subtle point, we make the following assumption: we query (and therefore model) sets of XML documents. We will call such a sef of documents an XML *data set*. XML elements in a data set can be partitioned according to their *types*: an XML element that has the form <*tag_name*>...</*tagname*> is said to be of type *tag_name*. Therefore, an XML data set can contain multiple elements of type *document*.

We model an XML data set D as a graph (noted G_D) as follows:

- For each element e in the XML data set, there is an internal node N_e in the graph G. Each such internal node N_e is labeled with a distinct system generated element ID, *elID*.
- For each data value v in the XML data set there is a corresponding leaf V_v. Each such leaf node V_v is labeled with the value v.
- The graph G_D contains two types of edges: attribute edges and content edges. They are built as follows:
  - If the element e_2 is directly nested within the element e_1, then graph G_D contains a content edge from node N_{e1} to node N_{e2}. This edge is labeled with the type of the element e_2.
  - If the value v is directly contained in the element e, then graph G_D contains a content edge from node N_e to node V_v. The edge is labeled with the empty string.
  - If the value v is the value of an attribute of the element e, then graph G_D contains an attribute edge between node N_e and node V_v. The edge is labeled with the attribute name.

Moreover, two types of data models can be considered: an *unordered* data model or an *ordered* data model. The ordered model enforces an additional total order on the set of outgoing content edges of each internal node, according to the order of the components of each element in the original XML file. For the sake of simplicity, we will consider only the unordered data model in this paper; in this data model the order of elements is ignored. If not stated otherwise, we will also ignore IDREFs. In our data model, IDREFs would be represented by an additional type of edges. Most of the techniques presented in this paper are not affected by the presence of IDREFs.

2.1.1 Example XML data

We use the following XML data set from the bibliographical domain to exemplify the proposed data model. It consists of three article elements in one XML document, called "bib.xml". The three article elements contain similar types of information (e.g. authors, title, year of publication of the article) but this information is organized differently in each case.

```
<document>
<article id="1">
        <author><name>Adam Dingle</name></author>
        <author><name>Peter Sturmh</name></author>
        <author><name>Li Zhang</name></author>
        <title>Analysis and Characterization of Large-Scale Web Server Access Pat
        <year>1999</year>
        <booktitle>World Wide Web Journal</booktitle>
</article>
<article  id="2" year="1999">
        <author name="A. Dingle" ></author>
        <author name="E. Levy" ></author>
        <author name="J. Song" ></author>
        <author name="D. Dias" ></author>
        <title>Design and Performance of a Web Server Accelerator</title>
        <booktitle> Proceedings of IEEE INFOCOM </booktitle>
</article>
<article id="3">
        @inproceedings{IMN97,
```

```
        author="Adam Dingle and Ed MacNair and Thao Nguyen",
        title="An Analysis of Web Server Performance",
        booktitle="Proceedings of the IEEE Global Telecommunications Conference (
        year=1999}
</article>
</document>
```

The graph modeling this XML data set is depicted in Figure 1. The attribute edges are marked in dashed lines; system generated elIDs for internal nodes are showed in bold italic.

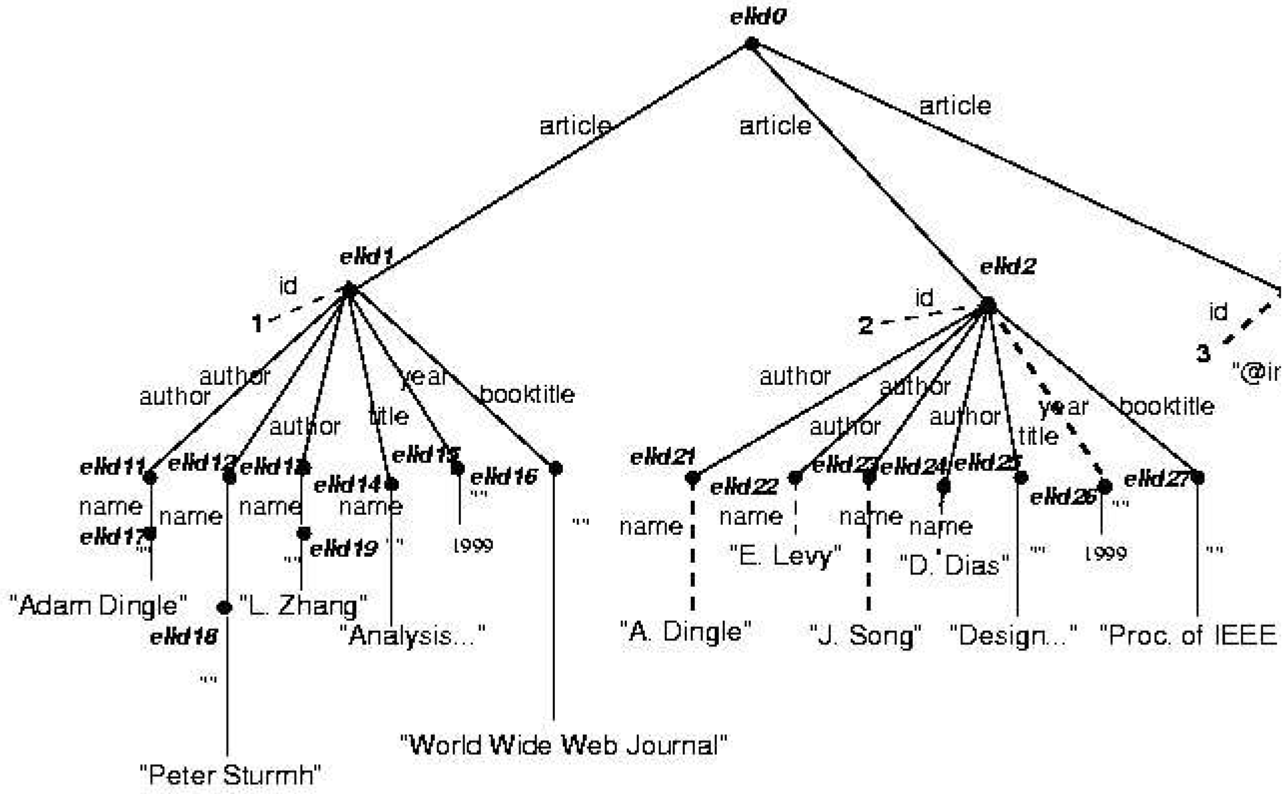


Figure 1: Data model corresponding to the XML example data.

2.2 The XML-QL Query Language

XML-QL queries specify declaratively the selection of data from multiple XML data inputs and the creation of an XML data output. A simple query in XML-QL has the following form:

```
where ( XML-pattern [ ELEMENT_AS $elem_var] )* IN fileName, ( predicate )*
construct  XML-pattern | variable
```

The **where** clause specifies how to filter data from the input XML data set; the **construct** clause specifies how to assemble the query results in XML. Syntactically, the **where** clause is composed of a set of XML patterns and a set of predicates. An XML-pattern is an XML element in which some of the items (tag names, data content, attribute names or attribute values) are eventually replaced with variables. A variable name is prefixed with a $ in order to distinguish it from a string value.

Similarly to an XML element, an XML-pattern can be modeled by a graph, with the major difference that some of the labels in the graph are now variables instead of constants. Let us denote as G_q the graph modeling the set of all XML-patterns in a query q.
Given an XML data set D, the result of the evaluation of an XML-QL query q on the input D is defined as follows. Each mapping from the graph G_q to the graph G_D which preserves the constant labels from the query induces a substitution of the variables in the query q on the set of constant values in the data graph G_D. For each such substitution which also satisfies the additional predicates, a new element, created by the instantiatiation of the XML-pattern in the **construct** clause, is added to the result.

An XML-pattern can be eventually followed by an *ELEMENT_AS $elem_var* clause. In this case, a mapping from the pattern's graph to the data graph will also define a substitution of the variable *$elem_var* on the set of internal nodes in the data graph (corresponding to elements in the original XML file). This variable can be projected in the *construct* clause; the result will then include the corresponding XML element.
Example 2.1 Let us consider the following query: "retrieve all articles written by Mr. Dingle in 1999 about the Web".

```
where <article><author><name>$N</name></author>
               <title>$T</title><year> 1999 </year>
      </article> ELEMENT_AS $E IN "bib.xml",
      $N like  *Dingle*, $T like  *web*
construct $E
```

There are three "query-to-data" mappings preserving the constants. These three mappings define the following substitutions:

*{$N/"Adam **Dingle**", $T/"Analysis and Characterization ... **Web** ...Performance", $E/elID1 }*
*{$N/"Peter Sturmh", $T/"Analysis and Characterization ... **Web** ... Performance", $E/elID1 }*
*{$N/"Li Zhang", $T/"Analysis and Characterization ... **Web** ... Performance", $E/elID1 }*

Of these three substitutions, only the first one matches the additional predicates. We note that in XML-QL the *like* predicate has the same semantics as in SQL, i.e. it tests if a certain string value matches a particular string regular path expressions. Hence, according to the semantics of the query language, the first XML element in our example is projected out in the result.

2.3 Extending XML-QL with Keyword Search

XML-QL provides a good solution to query XML data, assuming that a reasonable amount of knowledge about the structure of the data is given. In order to facilitate the case when no or limited knowledge about the structure of the data is given, XML-QL proposes (as all the other XML query languages) the use of regular path expressions. Unfortunately, regular path expressions are not the panacea for this problem, as the following example demonstrates.

Example 2.2 Let us assume that the user does not know if *"name"* is the name of a tag of one of the subelements of the author element or if it is the name of an attribute of this element; moreover, in the first case, the user might not know at which depth the tag *name* appears. The query is expressed in XML-QL as follows:

```
/* search for  "name" as a  subelement,  at  any depth */
where <article><author><*><name>$N</name></*></author>
               <title>$T</title><year> 1999 </year>
      </article> ELEMENT_AS $E IN "bib.xml",
      $N like  *Dingle*, $T like  *web*
```

```
construct $E
union
/* search for  "name" as an attribute  name, at any depth */
where <article><author><*><_ name=$N></_></*></author> <title>$T</title>
               <year> 1999 </year>
      </article>  ELEMENT_AS $E IN "bib.xml",
      $N like  *Dingle*, $T like  *web*
construct $E
```

It is easy to imagine the complexity of the query in the absence of *any* knowledge about the usage and the potential nesting of *article*, *title*, *year*, *name* and *author*. Such queries are both hard to write and expensive to evaluate.

To solve this problem, we propose to add a special predicate called *contains* to the XML-QL query language. The *contains* predicate tests the existence of a given word within an XML element. The *contains* predicate has four arguments: an *XML element variable*, a *word*, an integer expression limiting the *depth* at which the word is found within the element and a boolean expression over the set of constants *{tag_name, attribute_name, content, attribute_value}* imposing a constraint on the *location* of the word within the given element.

Given a data set *D* and a substitution of the variable *$E* into the set of internal nodes of the graph G_D, the predicate *contains(w,$E,d,loc)* evaluates to *true* iff *w* appears as a "word" in the set of strings labeling the edges and the leaves of the subtree of depth *d* rooted at *$E* in the graph G_D. Moreover, the occurrence of the word *w* within *$E* has to obey to the specified location (e.g. *tag_name* or *attribute_name*).

Example 2.3 The XML-QL query in the previous example can be simplified using a *contains* predicate. Formally, the idea is to search for all elements of type *article* having an *author* subelement and containing the strings "Dingle" below the *author* subelement and "1999" below the article itself. We limit the search to subtrees of depth 3 and we are interested in any kind of occurrences (i.e. the word can appear within a tag, an attribute name, within the content of an XML element or finally within the value to an attribute).

```
where   <article> <author> </author> ELEMENT_AS $A, <title>$T</title>
        </article> ELEMENT_AS $E IN "bib.xml",
        contains($A,`Dingle",3, any), $T like *web*, contains($E,"1999",3,any)
construct $E
```

In this example we used *any* as an alias for the expression *tag_name* OR *attribute_name* OR *content* OR *attribute_value*. According to the semantics of the *contains* predicate, the result of the evaluation of this query on the data set given previously contains the elements 1 and 2.

A relevant question is what is a "word" in this context. Intuitively, a word is a substring of a string value, with a separate identity and carrying a separate meaning. For example, a text document can be reasonably split into English words with separate meaning. However, in a more general context, it can be very difficult to define the notion of a "word". If the content of an XML element is not a fragment of text, but a string encoding some application domain information (e.g., the *bibtex* entry in the third article), the methodology of splitting PCDATA values into independent, meaningful "words" is application-dependent, too. In this example, the characters "@", "{" and "," can be used as separators. In our work, we assume that such application dependent procedures are given to the system. Another interesting question is whether there are normalization procedures (e.g. transform capital letters to non-capital letters) that can be applied to string values in order to increase the probability of matching the string constants in the query to values in the data. Again, if there are such application specific procedures, we assume that they are given to us. A related field of work for application-dependent splitting and normalization procedures is the domain of data cleaning; a framework describing the integration of such external procedures in the process of data cleaning is described in [GFSS00].

To better illustrate keyword-based search capabilities, we will give another example.

Example 2.4 Asume that we want to ask the same query, but in the absence of *any* knowledge of the structure of the XML data. In this case, we are looking for all article elements containing the words "Dingle", "web" and "1999" in subtrees of depth 3. In our extension of XML-QL this would be expressed as:

```
where <article> </article> ELEMENT_AS $E IN "bib.xml",
      contains($E,`Dingle",3, any), contains($E,"1999",3,any),
      contains($E,"web",3, any)
construct $E
```

This last query returns all three XML of our XML data set.

We conclude the section with the following remarks. First, keyword based search is a necessity in multiple situations: (a) if the XML data does not have structure (e.g as in our third article example), (b) if the structure is unknown to the user, and (c) if multiple heterogeneous XML sources need to be queried. Second, in order to be able to express all the range of queries, from fully structured ones to completely unstructured ones, and in order to allow a *global* optimization process of such queries, it is essential to support the two capabilities within the same query formalism.

3 Relational Support for Full-text Indexing

In this section we describe an extension of inverted files for full-text indexing. An extended inverted file can be used to implement keyword search (i.e., the *contains* predicates) and to find relevant XML data sources or XML elements in a distributed environment. Furthermore, we will show how inverted files can be stored in a relational database and discuss variants. How inverted files are used during query processing is detailed in Section 4.

3.1 Inverted Files

The classic index structure for keyword search is the inverted file [FBY92]. In the simplest form, an inverted file contains records of the following form:

<*word, document*>

meaning that the *word* can be found in the *document*. For our purposes, we need to extend this structure in the following ways:

1. We need to keep information in the granularity of elements rather than documents. For example, the word "Analysis" appears in a *title* element and in an *article* element in the example of Section 2.1.
2. We need to record whether a keyword is the name of a tag (e.g., *article*), the name of an attribute (e.g., *id*), a word in the value of an attribute, or a word in the element's data content. In a traditional information retrieval system, such a distinction is not required because such systems work in the granularity of documents; these systems are not geared towards XML, and they do not support *contains* predicates that need this information.
3. We need to record the *depth* at which the word appears for the first time within an element in order to

execute *contains* predicates that limit the depth. Again, traditional IR systems do not record this information because they ignore the structure of a document.

As a result, we propose to use inverted files that have records with the following structure:

<elID, word, depth, location>

where the *word*, *depth* and the *location* have the natural meaning. For the example of Section 2, the inverted file would contain, among others, the following records:

```
<"article",  elID1, 0, tag>
<"id", elID1, 1, attr>
...
<"name", elID1, 2, tag>
<"Adam", elID1, 2, value>
...
```

Note that it is possible to have multiple distinct records for the same word and ELID. For example, the word "year" might appear as a tag name at depth 1 within an article and as a value at depth 2 within an article with the title "The Year 2000 Bug."

A relevant question is how elIDs of elements are modeled and efficiently stored. In our system we model each elID as a record containing the URL of the belonging XML document, the starting and ending positions of the element within this document, and the *type* of the element. Furthermore, if the DTD of the document specifies that the element has an attribute that uniquely identifies that element (e.g., the *id* uniquely identifies an *article*), then the value of that attribute is also stored in an elID record. Rather than manipulating such complex elIDs, each elID record is given an internal key number and all the information of the element is stored in a separate table called the *elements* table. This way, we can work internally with much smaller elIDs and query processing becomes significantly faster. Similarly, we propose to store all information concerning individual XML documents (e.g., URL and information about the DTD) in a separate *documents* table. As a result, we obtain the following relational schema to store element elIDs and document information: (Keys are represented in bold face.)

*elements(**elID**, docid, start_pos, end_pos, type, id_val)*
*documents(**docid**, URL, ...)*

year

source	target	value
elid1	elid15	-
elid2	elid26	-

id

source	target	value
elid1	-	1
elid2	-	2
elid3	-	3

author

source	target	value
elid1	elid11	-
elid1	elid12	-
elid1	elid13	-
elid2	elid21	-
elid2	elid22	-
elid2	elid23	-
elid2	elid24	-

title

source	target	value
elid1	elid14	-
elid2	elid25	-

Figure 2: Binary Tables resulting from the XML example in section 2.1.1

3.2 Storing Inverted Files in a Relational Database

Inverted files can be implemented in many different ways. For our work, we chose to use a relational database system (RDBMS) because RDBMSs can handle large amounts of data and inverted lists obviously tend to become very large - larger in fact than the original data. In addition, as we will see, an RDBMS can also be used to execute the other parts of an XML-QL query so that a whole XML-QL query with keyword search can be processed by a single RDBMS without crossing process boundaries.

The natural way to store an inverted file in an RDBMS is to establish one table that stores the whole inverted file; this table would have four columns: *elID, word, depth,* and *location* as described in the previous subsection. Such a table, however, would be huge and this whole, huge table would have to be inspected for each query that involves a keyword search. It is therefore desirable to partition this table into several smaller tables so that each query can be processed by looking at a few small tables only. There are many ways to partition this table and the best way depends on the query workload and characteristics of the XML data. In this work, we propose the following heuristics which have proven to be quite effective in all our experiments:

- First, we partition by *word*. That is, for each keyword *w*, a separate table *w* is established with three attributes: *elID, depth, location*. To find all elements that contain the words "title" and "analysis" only the *title* and *analysis* tables would have to be inspected. Partitioning an inverted file in this way is not a novel idea; this is done by virtually all IR systems today [FBY92].
- Second, we further partition the individual *word* tables by the *type* of the elIDs. For instance, we would record all *article* elements that contain the keyword "Dingle" in a *Dingle-Article* table and all *author* elements that contain the keyword "name" in a separate *Name-Author* table. This partitioning is very useful for queries that specify the *scope* of the answer; an example is the second query of Section 2 which looks for all articles that have "Dingle" as one of the authors. To execute this query, we only need to inspect the *Dingle-Article* table. Partitioning the data in this way is novel and specific to our particular goal of processing XML queries and retrieving individual elements rather than whole documents.

As a result, we obtain the following relational schema to store inverted files: for each type (i.e. tag name) and for each word in the XML data set store a table

```
word-type(elID, depth, location)
```

Given the high number of possible *word-type* pairs, the inverted file will be stored in thousands, if not tens of thousands, of tables, but current RDBMSs are very well capable of managing such a large number of tables. If the number of tables is a concern, the partitioning can be limited to popular words; e.g., words that occur in more than 1000 elements.
We propose to store all tables sorted by elID (i.e., the number that internally represents the elID). Again, this will speed up query processing because finding all elements that contain two specific keywords can be achieved by *merging* the relevant tables for the two keywords. In most RDBMSs, such a sorting can be achieved by creating a clustered index on the elID column of the table. Furthermore, we propose to establish indexes on the *docid* and *type* columns of the *elements* table and on the *docid* and *URL* columns of the *documents* table.

3.3 Variants

As mentioned in the previous subsection, there are many different ways to partition the information of the inverted file in a relational database. For example, an alternative to our proposed scheme would be to partition by *word* and by *docid*. This scheme would be beneficial if most queries retrieve information from specific documents, rather than considering *all* documents. Ultimately, the best partitioning scheme depends on the query workload. However, we have had no problems with the partitioning scheme described in the previous section.

One way to speed up query processing is to materialize the intersection (or join) of two tables. For instance, many queries will ask for *articles* and involve the keywords "name," "author," and "title." As a consequence, it is beneficial to materialize the join of the corresponding *Name-Article, Author-Article,* and *Title-Article* tables into a *Name|Author|Title-Article*. Now, bibliographic queries can be processed directly using such a *Name|Author|Title-Article* table instead of the individual tables. The resulting relational schema is as follows:

Name|Author|Title-Article(elID, depth_Name, loc_Name, depth_Author, loc_Author, depth_Title, loc_Title)

All elIDs in this table refer to *article* elements. Note that, say, the word "name" may also appear in different contexts so that this materialization is indeed a useful prefiltering for bibliographic queries. For other queries, it might be useful to materialize the join of the *Name-Emp* and *Salary-Emp* tables. What exactly to materialize depends on the query workload.

A popular approach in the IR community is to ignore so-called *Stop* words in the inverted file. Stop words are words that appear very frequently in *all* or most documents; examples are the words "the" or "is." Stop words can be ignored in IR systems because IR systems try to find the most relevant documents that match a query and Stop words are not important in order to determine the relevance of a document. For our purposes, however, Stop words must not be ignored. For instance, words like "name" are likely to appear very frequently as element tags and they carry important information for the elements they appear in. Furthermore, we are interested in *all* answers that *fully* match a query, whereas IR systems are geared towards giving the ten or hundred documents that *best* match the query. For instance, counting the number of citations of works by Donald Knuth in all Computer Science papers is a valid query and involves finding *all* citations even if thousands exist.

Other approaches to improve query processing performance include the use of bitmaps and compression techniques [FBY92]. Principally, these techniques can be exploited in our context as well; however, so far we were not able to implement these techniques as part of our work since we relied on the capabilities of standard, off-the-shelf RDBMSs. Bitmap indexing techniques and database compression techniques are currently being integrated into several relational database systems [Joh99, IW94, WKHM98]; therefore, it should be possible to explore the usefulness of these features in future work.

4 Extended XML-QL Query Processing

We will now turn to a discussion of how XML-QL queries with *contains* predicates can be processed. We will first describe query processing in the first scenario of the introduction, in which the inverted file and all the XML data are stored or replicated in an RDBMS. After that, we will discuss the second scenario of the introduction.

4.1 Replicating the XML Data in an RDBMS

4.1.1 XML-QL Query Processing with an RDBMS

We will first revisit how XML-QL queries *without* keyword search can be implemented using an RDBMS. We will then describe the necessary extensions to integrate keyword search.

There are many different ways to store XML data in an RDBMS and to execute XML queries using and RDBMS; see, e.g., [DFS99, SGT+99, FK99. In this work, we will only consider the *binary table* approach which was presented in [FK99] because this approach shows good and robust performance. (Any other approach, however, could be used just as well for our purposes.) The basic idea of this approach is to establish a *binary table* for each *type* (i.e., tag name) that appears in the XML data. Each entry of a binary table `t` contains two elIDs, `source` and `target`. Such an entry denotes that `target` is a subelement of `source` (at depth 0) and that `target` is of type `t`. Alternatively, such an entry can denote that `source` has an attribute or subelement `t` which contains an IDREF to `target`. In this way the binary tables represent the structure of an XML document. Base values such as PCDATA or integers can be stored within the binary tables or in separate value tables. In this work, we stored values within the binary tables so that the binary table for all *articles* would have the following structure: (Here and in the following, we represent binary tables and their attributes by typewriter font so that they are not confused with the tables and attributes that store the inverted file. The attributes that build the key of the table are again represented in bold face.)

`article(`**`source`**`, `**`target`**`, value)`

`source` is an elID that references an *article*. `target` is an elID that references a subelement of an *article*; `target` is *null* if an *article* has no subelements. `value` is some (text) value which is part of the *article* (not part of a subelement); `value` is *null* if an *article* has no text value.

Details of the whole binary approach can be found in [FK99]. In particular, [FK99] explains how any kind of XML-QL query can be translated into an equivalent SQL query; including complex XML-QL queries that involve regular path expressions and/or wild cards. To give an example, Figure 2 shows some binary tables for the XML data set of section 2.1. The first XML-QL query of Section 2.2 could be executed by the following SQL query:

```
select art.source
from article art, author aut, name n, title t, year y
where art.target=aut.sources and art.target=t.source and art.target=y.source
  and aut.target=n.source and y.value=1999 and t.value like "web" and n.value lik
```

4.1.2 Keyword Search with an RDBMS

If the inverted file is stored by the same RDBMS that also stores the inverted file, then keyword search can be integrated in a straightforward way. If the query involves a *contains* predicate for a specific keyword, then the tables that store the entries of the inverted file for that keyword need to be joined as part of processing the whole query. This approach can again be best demonstrated by the means of examples.

Let’s consider a simple query searching for elements of type *article* which mention the word "Dingle". This query can be expressed in our extension of XML-QL as follows:

```
where   <article> </article> ELEMENT_AS $E IN "bib.xml",
        contains($E,`Dingle",3, any)
construct $E
```

To execute this query, our extended XML-QL query processor would translate it into the following simple SQL query:

```
select elID
from Dingle-article;
```

As another example, consider a query that asks for the names of all authors that have written a paper in 1999 mentioning the word "Web." This example demonstrates how keyword search and "structured" XML query processing coexist. In XML-QL, this query could be formulated as follows:

```
where   <article>
                <year> 1999 </year>
                <author> $A </author>
        </article> ELEMENT_AS $E IN "bib.xml", contains($E,"Web",4, any)
construct $A
```

This XML-QL query would be translated into the following SQL query:

```
select aut.target
from  article art,  year y, author aut, web-article c
where art.target=year.source and art.target=aut.source and art.target=c.elID
          and year.value="1999" and c.depth<=4
```

To execute this SQL query, the optimizer of the RDBMS would find a good order in which to join the individual tables. From a different point of view, the RDBMS interleaves keyword search (joins with the *web-article* table) with regular XML-QL query processing using the binary approach (joins with the `article`, `year`, and `author` tables). Of course, the RDBMS is not aware that it is doing keyword search and XML-QL query processing.

Keyword search and an inverted file might even be helpful in order to execute XML-QL queries without *contains* predicates. In particular, if the join of several tables that store the inverted file is materialized, this materialized join can be used as a prefilter in order to speed up the execution of the query. For instance, the last query can be rewritten into the equivalent query involving a *year|author|web-article* table (as a prefilter for the structured part) as follows:

```
select aut.target
from article art,  year y,  author a, web-article c, year|author|web-article A
where art.target=year.source and art.target=aut.source and art.target=c.elID and
         and c.depth<=4 and A.elID=art.target
```

Again, the optimizer of the RDBMS will decide when to carry out the join with the *year|author|web-article* table. If this table is small, then it is a good prefilter and the RDBMS will consider it early. Otherwise, it is a poor prefilter and the optimizer will not consider it until the end; in fact, the use of this table will increase the cost of the query in this case. Therefore, this kind of prefilter should only be used with care.

4.2 Distributed XML Query Processing

We now turn to a discussion of how to process XML-QL queries if the XML data can be indexed using an RDBMS, but the data cannot be stored in the RDBMS. Such a situation could arise on the Web, for example, if the owners of a Web site give permission to index the content, but users must visit the Web site in order to retrieve the data. In such a scenario the inverted file stored in the RDBMS is used to locate relevant XML data sources. The full query result must be computed by a mediator. The mediator uses the RDBMS to query the inverted file and accesses the XML data sources to compute the full query results. We differentiate two cases: (1) the data sources have no query capabilities; and (2) the data sources have some query capabilities.

4.2.1 XML Data Sources Without Query Capabilities

If the data sources have no query capabilities, then XML-QL queries are executed in the following way:

1. **Prefilter**: use the inverted file stored in the RDBMS to find the relevant documents and/or elIDs.
2. **Retrieve**: get the relevant documents (or elements) from the data sources.
3. **Execute**: extract the query results from the retrieved documents (or elements).

Prefiltering is the most interesting step. It tries to narrow down the search as much as possible by considering all tag names, attribute names, and keywords that appear in the query. For prefiltering, the inverted file stored in the RDBMS is useful even if the query does not involve any *contains* predicates. How prefiltering works can again best be explained by looking at an example.

Let us consider a query that asks for the names of all authors of a article with the keyword "analysis" in the title. In XML-QL, this query can be written as follows:

```
where  <article>
              <author> <name> $n </name> </author>
              <title> $t </title>
       </article> IN "bib.xml",  $t like *analysis*
construct $n
```

We can use the inverted file to get the following information:

- get the elIDs of all *article* elements that have an "author" subelement at depth 1;
- get the elIDs of all *article* elements that have a "name" subelement at depth 2;
- get the elIDs of all *article* elements that have a "title" subelement at depth 1;
- get the elIDs of all *article* elements that contain the word "analysis" at depth at most 2.

Clearly, we are only interested in articles that meet all four criteria; let us call this set of *articles* *C*. *C* contains all *articles* which are relevant, but it might also contain some *articles* that do not match the query; for instance, an article that contains the word "analysis" somewhere, but not in its *title*. To narrow down the search even further, we can use the inverted file to get all *interesting title* and *author* elements; i.e.,

- get the elIDs of all *author* elements that have a "name" subelement at depth 1;
- get the elIDs of all *title* elements that contain the word "analysis" at depth 1.
- from the start and end positions of the elIDs stored in the *elements* table, we can now infer which of the *article* elements of *C* have *title* and *author* subelements belonging to the previous sets; this is the result of prefiltering.

The whole prefiltering can be done with a single SQL query. We do not show this query for brevity. In this example, prefiltering proves to be very good. That is, prefiltering will only return the elIDs of *article* elements which match the query. In general, prefiltering is not always perfect. For example, predicates like *price* < 1000 cannot be evaluated using the inverted file. Furthermore prefiltering with an inverted file cannot be perfect, if the XML data contains IDREFs. In this case, the subelement test with the start and end positions stored in the *elements* tables is not applicable and prefiltering must stop after the first step and return the elIDs of all *article* elements in *C*. Another situation in which prefiltering is not perfect is if traditional inverted files or inverted files with less information are used instead of the full-fledged, extended inverted files proposed in Section 3. In any case, it is important that prefiltering finds a *superset* of all elements that match the query.

Independent of the quality of the prefiltering step, the mediator must access the XML data sources. In our example, for instance, the mediator must access the data sources in order to construct the names of the authors. The implementation of the retrieval step depends on the interfaces of the XML data sources. If the data sources support the retrieval of individual elements and their subelements, then this feature should be exploited. Otherwise, the entire documents that contain relevant elements must be retrieved.

The query results can be produced in the mediator in a straight-forward way: while parsing the retrieved documents (or elements), the mediator can check whether they match the query (if prefiltering is not perfect) and construct the query results. If the query is complex, many large documents are retrieved, and matching is complicated, the documents can also be (temporarily) loaded into the RDBMS as described in Section 4.1; in this case, it is also possible to use the inverted file stored by the RDBMS in order to evaluate *contains* predicates. If the retrieved XML data contains IDREFs, the mediator might need to follow these IDREFs and retrieve further documents as part of query execution.

4.2.2 XML Data Sources With Query Capabilities

As an alternative to retrieving and generating query results in the mediator, it is also possible to push parts of the query down to the data sources, if the data sources have query capabilities. In this case, query processing is carried out in the following three steps:

1. **Prefiltering**: find all relevant documents and elIDs as described in the previous subsection.
2. **Push Down**: pass elIDs to the data sources and let the data sources execute the whole or parts of the query on these elIDs.
3. **Refinement**: refine the results returned by the data sources if the data sources could not execute the whole query.

Step 2 can be executed in many different ways; in the database literature this step has been called *bind* or *dependent join* and it has been studied extensively. We will not go into details here and refer the interested reader to the relevant database literature; e.g. [FLMS99].

5 Experiments

We implemented a prototype XML-QL query processor with keyword search on top of an off-the-shelf RDBMS. In this section, we will present the results of initial performance experiments conducted with our prototype. We will only report on experiments performed in a a scenario in which all the XML data (in addition to the inverted file) is replicated in the RDBMS.

5.1 Experimental Environment

We constructed a database with all of Shakespeare's plays in XML format. The DTD and the XML representation of the plays are available from the Oasis Web site (www.oasis.org). We also experimented with synthetic XML documents (a random XML generator) and various kinds of DTDs, but we will not present the results of these experiments for brevity.

All our test programs were written in Java, using JDK1.2. The RDBMS server was Oracle8i running on a PC with WindowsNT and a 2Gb disk. The client programs ran on a separate PC, equipped with a Pentium at 400Mhz, running under Redhat Linux 6.2. We used the XML4J parser available at www.alphaworks.ibm.com, version 2.0.15. This is a DOM parser that constructs an in-memory parse tree for the whole document. For the Shakespeare data which consists of fairly small documents that fit into main memory, this parser worked very well. If large documents whose size exceeds the main memory need to be parsed, an event-based parser would work better.

5.2 Generating the Inverted File and Relational Database

The indexing process proceeds in several steps:

1. We parse the XML document into an in-memory structure, as described in the previous subsection.
2. From the parse tree, we construct records of the inverted file. We store these records in a (temporary) file on the file system.
3. We bulkload the file with the records of the inverted file into one big "contains" table of the RDBMS.
4. We partition the "contains" table into tables for each *word* and *type* of elID using SQL queries, as described in Section 3.

The most expensive step is the second step. In this step, we need to check each word (or attribute or tag name) of the XML document and generate zero, one, or several records. First, we need to check whether the word already exists in the inverted file or is a *new* keyword which requires special handling. Then, we need to check whether this word is the first occurance of this word in the current element (and its father element, grandfather element, etc.); only if it is the first occurance, a record is generated. Table 5.3 summarizes the size and loading time of all of Shakespeare's plays with our prototype.

Size of orig. XML file	Number of distinct words	Size of relational database	Loading time
7.7Mb	30468	90Mb	27 min.

We note that the size of the relational database (including *binary tables, inverted file*, and database indexes) is about ten times the size of the original XML. The main reason for this is that we construct inverted files in the granularity of XML elements, rather than documents. Building inverted files in such a fine granularity is a must for combined keyword search and XML query processing -- there is nothing we can do about that. We are aware, however, that such an explosion in the data volume might be unacceptable for some applications. We therefore plan to investigate special database compression techniques and the use of *approximate inverted files* as one very important avenue for future work.

The total loading time is also quite high. Roughly half of this time is spent writing the document content into temporary files. The rest of the processing time is spent loading the data into the RDBMS and constructing the appropriate indexes. We believe that by fine-tuning the DBMS and by judicious distribution of processing between the DBMS and the file system, this time can still be reduced.

5.3 Query Performance

We studied two different queries. The first query asks for all *lines* by the *speaker Iago* which contain the word *love*. The second query asks for all *scenes* that contain the word *the*. We implement each query in three different variants:

- **Structured.** The query exploits the full XML structure; i.e., the query involves no *contains* predicate. This variant simulates an expert user.

- **Partially structured.** The query exploits some structure; i.e., plays have scenes and scenes *somehow* involve lines and speakers. This variant models a user who has partial knowledge of the structure of the data.
- **Unstructured.** The query has no structure; it is entirely composed of *contains* predicates. This models a user with absolutely no knowledge of the structure of the data.

The first query is expressed in the three variants as follows.

1. Structured query:

```
where <play> <act> <scene>
                     <speaker>  Iago </speaker>
                    < line>  $L </line>  ELEMENT_AS $E IN "bib.xml",
                 </scene> </act> </play,>  $L like  *love*
construct $E;
```

2. Partially structured query:

```
where <play> <act>
                         <scene>  </scene>  ELEMENT_AS $E IN "bib.xml",
         </act> </play,>
            contains($E, "love", 5, content), contains($E,"Iago", 5, content),
            contains($E, "speaker", 5, tag_name), contains($E, "line",
5, tag_name)
construct $E;
```

2. Unstructured query:

```
where <_> </_ >  ELEMENT_AS $E IN "bib.xml",
            contains($E, "love", 7, content), contains($E,"Iago", 7,content),
            contains($E, "speaker", 7, any), contains($E, "line", 7,any),
            contains($E, "play",7, tag_name), contains($E, "act", 7,any)
construct $E;
```

The table below shows the running times and the query results produced by each variant. We observe that each query can be executed within seconds. The *partially structured* variants are somewhat faster than the *structured* variants which in turn are faster than the *unstructured* variants. The *precisest* query results can obviously be achieved with the *structured* variant. For the *partially structured* and *unstructured* variants, the user must issue further (more structured) queries in order to get the right results. For example, the *partially structured* variant of the first query returns results in the granularity of *scenes*; to get the right *lines* within these scenes, the user must refine the query, thereby considering the structure of the *scenes* returned in the first, partially structured attempt.

	Query 1		Query 2	
	Running Time	Query Result	Running Time	Query Result
Structured	3 secs	28 lines	7 secs	208 scenes
Part. Structured	1.5 secs	19 scenes	2.5 secs	1108 scenes
Unstructured	4 secs	56 elements	10 secs	10909 elements

To conclude, the best results can of course be achieved by expert users that fully know the structure of the data. In these experiments, however, *contains* predicates are very useful to support novice users. (As mentioned in the introduction, there are also reasons for experts to use *contains* predicates.) Novice users will refine their queries stepwise, add structure, and thus get more specific results. This approach is very well doable because each step only takes seconds on a simple PC (subseconds on a powerful multi-processor machine). Of course, we do not expect novice users to able to speak XML-QL (or any other XML query language); users will be able to use graphical query interfaces that generate XML-QL queries with *contains* predicates.

6 Related Work

Both "structured queries" and "keyword search" have extensively been studied in the database and information retrieval literature. Specific work on XML query processing is reported in [DFS99, SGT+99, Wid99], and information retrieval techniques such as those used in current Web search engines can be used for XML just as well as for HTML or any other text data. What makes our work different is that we show how keyword search can be integrated into (structured) query processing and why this works particularly well for XML.

As a starting point, we used the XML-QL query language [DFF+99], and extended it to integrate keyword search. Many other query languages for XML exist; e.g., XQL [XQL98] or XGL [CCD+99]. Most proposals for XML query languages have been presented in [XML98]. Also, XSL has simple query capabilities [XSL99]. Furthermore, several languages for semi-structured data have been developed; e.g., StruQL [BDHS96]. All of these languages support features like regular path expressions in order to search for patterns and structures from XML data - keyword search, however, is not supported. One exception is Lorel [Wid99], which has recently been extended by this feature in a similar way as we propose for XML-QL.

There are a number of companies that have products for storing and querying XML; examples are Excelon, Oracle's XML Developer's Kit[XDK], and Tamino. None of these products, however, supports keyword search. Oracle's XDK allows for full-text indexing of XML documents if they are stored as flat text, i.e. there is no way of using *both* a structured query language and the text index. Recently, a few start-ups have started working on query processing and keyword search for XML. One example is XDEX, a branch of Sequoia Software. These companies, however, have not yet launched any products for this purpose and neither have they published details of their approaches.

7 Conclusion

We showed how an existing XML query language can be extended in order to support keyword search. Furthermore, we described how such an extended XML query language can be implemented. The most important data structure needed for keyword search is the inverted file. We gave the necessary extensions of inverted files for XML query processing and showed how inverted files can be stored and queried using a relational database system. The techniques described in this paper can easily be implemented; the implementation becomes even easier if in addition to the inverted files, the XML data itself can be replicated in the relational database system.

Combining keyword search and regular (structured) query processing is definitely useful. Among others, a system that supports both allows users that have no or only partial knowledge of the structure of the XML data to ask and refine queries. Our experiments showed that keyword search and overall XML query processing can be carried out very efficiently. Typically, the more structure is known, the faster a query of a user will be executed; however, totally unstructured queries can be executed very fast, too. Also, the more structure is known, the higher is the quality of the query results. If little structure is known too many results will be produced (high recall, low precision), but again, the first results returned by a completely unstructured query can be used to refine the query in order to get better results.

The flip side of our proposed approach is that extended inverted files can become very large; it is not unusual that they are a factor of ten or even more larger than the original data. As a consequence, our most important goal for future work is to investigate variants of inverted files which are significantly smaller and show (almost) as good performance.

References

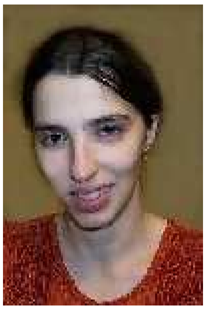
BDHS96 Peter Buneman, Susan B. Davidson, Gerd G. Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data (electronic version). In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 505-516, 1996.
CCD+99 Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A graphical language for querying and restructuring XML documents (electronic version). In *Proc. of the Int. World Wide Web Conference (WWW)*, volume 31(11-16), pages 1171-1187, 1999.
DFF+99 Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query language for XML (electronic version). In *Proc. of the Int. World Wide Web Conference (WWW)*, volume 31(11-16), pages 1155-1169, 1999.
DFS99 Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED (electronic version). In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 431-442, 1999.
Exc99 Excelon from ODI: http://www.odi.com/excelon.
FBY92 William B. Frakes and Ricardo A. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.
FK99 Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS (extended version). In *IEEE Data Engineering Bulletin*, volume 22(3), pages 27-34, 1999.
FLMS99 Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns (electronic version). In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 311-322, 1999.
GFSS00 H. Galhardas, D. Florescu, D. Shasha, and E.Simon. AJAX: An extensible data cleaning tool (electronic version). In *Proc. of ACM SIGMOD Conf. on Management of Data*, 2000.
IW94 Balakrishna R. Iyer and David Wilhite. Data compression support in databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 695-704, 1994.
Joh99 Theodore Johnson. Performance measurements of compressed bitmap indices. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 278-289, 1999.
SGT+99 Jayavel Shanmugasundaram, H. Gang, Kristin Tufte, Chun Zhang, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities (electronic version). In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 302-314, 1999.
Tam99 Tamino from SoftwareAG: http://www.softwareag.com/tamino.
Wid99 Jennifer Widom. Data management for XML: Research directions (electronic version). In *IEEE Data Engineering Bulletin*, volume 22(3), pages 44-52, 1999.
WKHM98 T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases, nov 1998. Submitted for publication.
XDK Oracle's XML Development Kit (XDK).
XML98 Query Languages - position papers: http://www.w3.org/TandS/QL/QL98/pp.html.

XML99 XML Index: http://www.xmlindex.com/index.html
XQL98 The XQL query language: http://www.w3.org/TandS/QL/QL98/pp/xql.html.
XSL99 The Extensible Stylesheet Language (XSL): http://www.w3.org/Style/XSL.

Vitae

Daniela Florescu received her Ph.D. in 1996, on "Search Spaces for object oriented query optimization". She is now a researcher at INRIA Rocquencourt, in the Caravel project. Dr. Florescu is among the authors of the XML-QL query language and the main designer of the Strudel web-site management system. Her current research interests include XML technologies (query languages, storage, query optimization), static query optimization, data-intensive web site management, and data cleaning. Daniela Florescu is also a member of the W3C working group on XML query languages (homepage).

Donald Kossmann received BSc and MSc degrees in 1989 and 1991 from the University of Karlsruhe (Germany) and a Ph. D. in Computer Science in 1995 from the Technical University of Aachen (Germany). From 1995 to 1996, he was a Research Associate at the University Maryland, College Park. Since 1996, he is an Assistant Professor for Computer Science at the University of Passau (Germany). His research is focussed on distributed and object-oriented database systems (homepage).



Ioana Manolescu received her MSc in 1998, from Ecole Normale Superieure, in Paris, and is now a Ph. D. student at INRIA Rocquencourt, in the Caravel project; her topic is "Query Optimization for Semistructured Data". She is also interested in XML schema extraction for storage optimization, and query optimization for data integration systems. Together with Daniela Florescu and Donald Kossmann, she is currently working on a new system that allows a relational data integration engine to seamlessly integrate XML documents (homepage).