On the Use of a Relational Database Management System for XML Information Retrieval*

Chun Zhang Qiong Luo David DeWitt Jeffrey Naughton Feng Tian

Department of Computer Sciences
University of Wisconsin, Madison
{czhang,qiongluo,dewitt,naughton,ftian}@cs.wisc.edu

Abstract

The growing number of XML files accessible on the Internet suggests that in the future, searching for XML files will become as important as searching for HTML files is today. XML information retrieval differs from HTML or plain text document retrieval in that the XML allows queries on the structure as well as the content of the documents. Perhaps the most logical way to build such a system is to follow the road the SGML search community has taken, and to build a special-purpose inverted-list index engine. In this paper, we investigate a different approach, that of using a commercial relational database system to support XML information retrieval. We show that using a relational database system offers some advantages that are not offered by traditional IR systems, including the standard features of query optimization, concurrency control, and recovery, but also greater flexibility and extensibility. The question, of course, is whether these benefits come at the expense of good performance. To begin to answer this question, we compared the performance of two commercial RDBMSs with that of our own inverted list XML information retrieval engine. Our study shows that while in general the database systems are not well tuned for IR queries, under certain conditions they can outperform the IR system. Our results and analysis further suggest that with additional research into techniques for supporting IR workloads, RDBMSs could become a viable alternative to special purpose inverted list systems for supporting XML information retrieval. Category: Research.

1 Introduction

Today's WWW contains vast amounts of data, and the utility of this data depends more and more on search engine technology that enables users to find what they are looking for. To date, search engine technology has relied upon implementations in custom systems that resemble classical information retrieval (IR) systems. In this paper we explore the question of whether that is the best approach — specifically, we investigate the use of a general purpose RDBMS as the storage and computational engine for XML data retrieval.

*Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908.

There are a number of reasons why this question is relevant. First, the WWW environment does not really resemble the classical information retrieval environment very closely. One of the main changes occuring on the web is a move toward more highly structured data, through the advent of XML [XML98]. This highly structured data enables far more complex and precise searches than are possible over traditional unstructured text documents. Other important differences are that the web is dynamic, and that the scale is simply enormous both in terms of data size and numbers of user queries.

These differences between the web and classical IR environments pose demands on search engines that are similar to those met by RDBMSs. The SQL query language can express powerful and complex queries; and 30 years of research and development has produced systems that have superb scalability, concurrency control, query optimization, and recovery; furthermore, there is a huge industry of supporting tools and applications for using RDBMSs.

Note that this work is different from work that studies using an RDBMS to store XML documents [SGT+99, FK99, TDC00]. Here we are not trying to store the documents themselves, rather, we are using the RDBMS as an engine upon which to build an index for these documents. This is a different problem; storing XML in relational database systems deals with mapping semi-structured data to a structured relational system. To the contrary, the indices on XML data sets are highly structured. So the questions to be addressed here include (a) what is the performance impact of this decision, and (b) what benefits might arise from using an RDBMS for this task.

Our technique transforms the inverted index, a widely used IR index, into a set of relational tables, and translates IR queries into SQL queries. Not only is the translation very simple, but also we found that, due to the flexibility of an RDBMS, we obtain a number of advantages that are not provided by a traditional IR system. These advantages include the ability to express complex searches involving joins, the ability to query based both on the XML data content and also other information stored in the RDBMS, and the ease with which the RDBMS-based system can be extended to handle query types that were not anticipated during the design of the original system.

Of course, all these benefits are irrelevant if the RDBMS performance is not satisfactory. For this reason, we conducted a performance study to compare the query performance with database support versus the query performance in a special purpose IR system. We used two commercial database systems in the study, and built our own IR system. Our goal was to investigate the potentials and weaknesses of using an RDBMS for information retrieval, and perhaps to gain insights into better database techniques to improve performance on IR workloads.

Our results show that in general, performance of RDBMSs on the IR workload is not very good, although under certain conditions, the RDBMS approach can outperform the IR system. We discuss the observations we made during our performance study and we suggest areas where an RDBMS can be improved to better support the IR workloads.

We begin in Section 2.1 with a discussion of related work. In Section 2.2, we introduce the IR inverted index structure that supports boolean, proximity, ranking and containment queries and the retrieval algorithms used to process these queries. In Section 3, we describe the relational schema counterpart of the IR inverted index, and show how IR queries can be translated into SQL queries.

The performance study is detailed in Section 4. We present our conclusions in Section 5.

2 Background and Related Work

2.1 Related Work

Although the fields of IR and database systems have largely gone their own separate ways in the past, work has been done on integrating information retrieval, especially text searching, with database systems. Examples of integrating text search with relational, object-relational, or object-oriented databases include [BCK+94, YA94, DM97]. Commercial examples include DB2 Text Extender and Oracle ConText Cartridge. An example of integrating text search with semi-structured databases is Lore [MAG+97], in which a simplified version of IR-style text index is used to locate strings containing specific words or groups of words [MWA+98].

There are two main approaches to integrating text retrieval with DBMSs in the literature. The first is a loosely coupled integration that keeps an IR system as an external component. The second is a tightly coupled integration that builds text searching functionality inside a database system. The former has performance problems mainly due to latency caused by crossing system boundaries, while the latter requires significant modification to the database system. In both cases, query optimization is an issue.

Our approach could be viewed as building IR functionality in an application *on top of* a database system. With this approach, there is no need modifying the database system, no external support is necessary, and there is no latency related to coordinating more than one subsystem. It leverages all the advantages that a database system has to offer, and all queries are optimized by the database system.

The advent of SGML [SGML86] triggered much research on integrating content and structure in text retrieval, including [BN96, AFL+94, BCK+94, M90, SAZ95]. Work on containment queries (Section 2.2) can be found in [CCB95a, CCB95b, DST96], and [CCB95a] proposes an algebra for containment queries. Our work on containment queries differs from the previous work in that, since we target at XML rather than SGML data retrieval, and XML elements are strictly nested, we are not concerned with overlapped extents, nor with reduction functions on inverted lists containing overlapped extents. Most significantly, our work does not focus on the development of containment algorithms; rather, it focuses on using an RDBMS to implement the algorithms.

Commercial search engines that combine content and structure in the retrieval have already started to appear. GoXML(http://www.goxml.com) is an example.

2.2 Inverted Index

The inverted index that we consider supports a superset of queries supported by traditional IR systems. These include boolean queries (so-called keyword searches), proximity and ranking queries, as well as *containment* queries—queries that restrict the search of keywords within the context of structural elements. For instance, the set of XML Shakespeare plays can be indexed, including

1. Find books with title "AltaVista Search Revolution".
2. Find the section with the last subsection of which containing the keyword "multimedia".
3. Find all citations in the article titled "Web Databases".
4. Find the first author of the book titled "Modern Information Retrieval".
5. In the speech spoken by "Antonio", find the line that contains "merchandise".
6. Find authors containing "donald" followed by "knuth" within 2 words.

Figure 1: Sample queries that use containment

all the markup tags, then one can query the <title> of plays containing a <line> that contains "caesar". Here both <title> and <line> are markup tags (called *elements*), and "caesar" is a text word. Figure 1 lists some sample queries that use containment.

The classic inverted list data structure [K73, SM83] records various information about a word, such as which document the word appears in and its position in the document. This type of structure is similar to the indices at the ends of books. In an IR system, a collection of inverted lists is stored in a file called an *inverted file*. One or more inverted files form one component of an inverted index, and one or more *lexicons* form the other component. A *lexicon* is a collection of indexed words or phrases plus links to inverted lists that reside in an inverted file. Multiple lexicons and inverted files can exist in a system. In general, a lexicon is small, accessed very often, and can be kept in main memory. An inverted file, on the other hand, is large and must stay on secondary storage.

There are variations of inverted index data structures. We used a simple one for our study. First we give some definitions that will be used throughout the paper. An *index term* is either an element or a text word. For conciseness, we use *index term* and *term* interchangably. A *position* is the location of a term in a document. For an element, the position is given by a begin and end word number pair; for a text word, the position is given by a word number. A *posting* is a pair of a document number and a position. Thus a posting pinpoints a term in a collection of documents. An inverted file is also called a *postings file*. The *frequency* of a term is the number of occurrences of the term in the whole dataset.

Figure 2(a) shows what a simple inverted index for XML files looks like conceptually. It has two lexicons, one for elements and one for text words. The contents of a pair of angle brackets is a posting, and the number, or number pair, following a semicolon in a posting is a position. For example, the "<book>" element spans in document 1 from word number 19 to 27 and occurs a second time in the document from word number 28 to 36. It also appears in document 2 from word number 1 to 9. Similarly, "java" appears in document 1 at word numbers 21 and 30 and in document 2 at word number 4.

Although this inverted data structure is simple, it is able to support keyword and boolean queries, ranking queries, proximity queries, and more importantly, containment queries. Proximity and containment relationships among terms are captured in the postings. In the example in Figure 2(a), we can see that the "<book>" element in document 1 at position $(19, 27)$ contains the "<title>" element in document 1 at position $(20, 23)$, as they are in the same document, and the first position

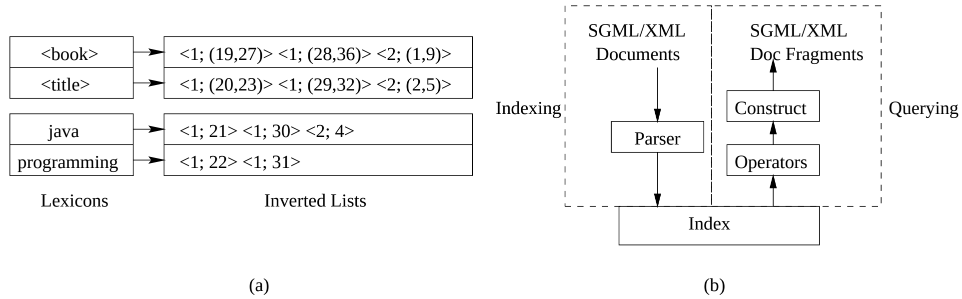

Figure 2: (a) A sample fragment of an inverted index (b) Building an inverted index and using the inverted index to answer queries

nests the second. We can also see that the title contains both "java" and "programming" as $(20, 23)$ nests both 21 and 22. In addition, the title is exactly "java programming".

The two paths, building an inverted index, and using the inverted index to answer queries, are depicted in Figure 2(b). Index terms are identified and postings are created during indexing. When evaluating queries, inverted lists are retrieved from the index and operators are invoked on them to produce intermediate results. Usually these results are not to be consumed by the end user, and some additional steps need to be performed. For example, it might be necessary to retrieve the original documents in order to construct fragments of it to return to the user.

2.3 IR Retrieval Algorithms

For the efficiency of query operations, all inverted lists are stored in sorted order. For the inverted index structure described in the previous section, a list is sorted first in increasing order of document number, then in increasing order of positions. This data structure makes operations on inverted lists very much resemble the merge phase of a sort-merge join in a database system. Let's look at the *containment* operation, which takes two terms as operands, and finds all occurrences of one term that contain the other.

func CONTAIN_FUNC (T_1, T_2) {
  Retrieve inverted lists L_1, L_2 for terms T_1, T_2;
  Prepare output inverted lists OL_1, OL_2;
  ptr_1 := head of L_1, ptr_2 := head of L_2;
  while ($ptr_1 \neq$ end of L_1 **and** $ptr_2 \neq$ end of L_2) {
    P_1 := posting pointed to by ptr_1, P_2 := posting pointed to by ptr_2;
    D_1 := document number in P_1, D_2 := document number in P_2;
    POS_1 := position in P_1, POS_2 := position in P_2;
    if ($D_1 = D_2$ and POS_1 **nests** POS_2) { /* *nests* has intuitive meaning */
      add P_1 to OL_1, add P_2 to OL_2;
      increment ptr_1 and ptr_2;
    }

```
      else if (D1 > D2) increment ptr2;
      else increment ptr1;
    }
    Return OL1, OL2;
  }
```

This code merges two operations into one. It finds both T1's that contain T2 and T2's that are contained in T1. Frequently we would like to get only one of these. For this purpose we use two operators: CONTAINS and CONTAINEDIN, which can be built on top of CONTAIN_FUNC. The former returns all *containers*, and the latter returns all *containees*.

Next we describe how two powerful and popular operations, *distance* (e.g. DISTANCE("java", "programming")$\leq$1) and *is* (e.g., title IS "java programming") are processed. But before that, let's define two operations on positions. First, a *cover* of N postions is a position that nests all of the positions and spans the minimum region. For instance, a cover of positions $(19, 21)$, 23, and $(22, 27)$ would be $(19, 27)$. Second, a postion $P1$ *tightly nests* position $P2$ iff $P1$ nests $P2$ and does not nest anything else. Thus, $(19, 22)$ tightly nests $(20, 21)$, but $(19, 23)$ doesn't.

The *distance* operation takes two text words, an operator such as '<', and a numeric value, and produces an inverted list. The *is* operation takes an element, a string of text words, and find occurrences of that element that exactly contain the string of text words.

```
  func DISTANCE_FUNC (T1, T2, op, value) {
    Retrieve inverted lists L1, L2 for terms T1, T2;
    Prepare output inverted list OL;
    ptr1 := head of L1, ptr2 := head of L2;
    while (ptr1 ≠ end of L1 and ptr2 ≠ end of L2) {
      P1 := posting pointed to by ptr1, P2 := posting pointed to by ptr2;
      D1 := document number in P1, D2 := document number in P2;
      POS1 := position in P1, POS2 := position in P2;
      if (D1 = D2 and op(POS1, POS2, value)=true) {
        OutPOS := cover (POS1, POS2);
        add OutPOS to OL;
        increment either or both of ptr1, ptr2 depending on op, value;
      }
      else if (D1 > D2) increment ptr2;
      else increment ptr1;
    }
    Return OL;
  }
  func IS_FUNC (Elem, T1, ..., TK ) {
    Retrieve inverted lists L0, L1, ..., LK for terms Elem, T1, ..., TK;
    Prepare output inverted list OL;
    ptri := head of Li (0 ≤ i ≤ K);
```

1. Find books with title "AltaVista Search Revolution".

book CONTAINS (title IS "AltaVista Search Revolution")

2. Find the section in which the last subsection contains the keyword "multimedia".

section CONTAINS (section[-1] CONTAINS "multimedia")

3. Find all citations in the article titled "Web Databases".

cite CONTAINEDIN (article CONTAINS (title IS "Web Databases"))

4. Find the first author of the book titled "Modern Information Retrieval".

author[1] CONTAINEDIN (book CONTAINS (title IS "Modern Information Retrieval"))

5. In the speech spoken by "Antonio", find the line that contains "merchandise".

line CONTAINS ("merchandise" CONTAINEDIN (speech CONTAINS (speaker IS "Antonio")))

6. Find authors containing "donald" followed by "knuth" within 2 words.

author CONTAINS (DISTANCE ("Knuth", "Donald") <= 2)

Figure 3: Sample queries using operators described

while ($ptr_i \neq$ end of $L_i (0 \leq i \leq K)$) {
  P_i := posting pointed to by $ptr_i (0 \leq i \leq K)$;
  D_i := document number in P_i $(0 \leq i \leq K)$;
  POS_i := position in P_i $(0 \leq i \leq K)$;
  if ($D_0 = ... = D_K$ **and** POS_i's are adjacent $(1 \leq i \leq K)$
      and POS_0 **tightly_nests** (**cover** $(POS_1, ..., POS_K)$)) {
    add POS_0 to OL;
    increment ptrs;
  }
  else increment ptrs pointing to smallest D_i's;
}
}

In addition to the operations above, there are others such as the familiar boolean AND, OR, and NOT. The processing of these operations requires intersection, union, and complement, respectively, of the inverted lists. Due to space constraints, we do not elaborate on them here. One thing to note is that they should be interpreted in the context of a containment operator, for example, <author> CONTAINS ("java" OR "programming"). When there is no explicit containment, an implicit one involving the root of an XML document is assumed.

Most of the operations have inverted lists as their inputs and outputs. Some take inverted lists and produce postings. These include the retrieval of specific positions such as the "first", "second", etc. occurrence of a term.

With these operations and in light of the algebra in [CCB95a], we can now express the sample queries in Figure 3. We call the last two queries the "Antonio" query and the "Knuth" query. We express these queries in SQL (Section 3.1) and show their performance in Section 4.3.

3 Information Retrieval on RDB

3.1 Schema for Inverted Index

We now turn the index structure described in Section 2.2 into a relational schema. We store the postings in two tables:

ELEMENTS (`integer termno, integer docno, integer beginno, integer endno`)

TEXTS (`integer termno, integer docno, integer wordno`)

The ELEMENTS table stores postings for elements, while the TEXTS table stores postings for text words. The column `termno` uniquely identifies a term. Note that it is not a key.

There are two ways to handle the lexicon. One alternative is to turn it into a relational table:

LEXICON (`varchar term, integer termno, integer frequency, ...`)

where both the `term` column and the `termno` column are keys, thus there is one row per distinct term. The column `frequency` and others may be used to support ranking or query optimization. The other alternative is to keep the lexicon external to the RDBMS in the same structure as is used in an IR system, and keep it memory resident. After the `termno`s are looked up in the lexicon, they are used for generating the SQL translation of IR queries. We used the second alternative for our experiments (Section 4). For brevity and purposes of illustration, when we speak of predicates of the form: `ELEMENTS.term = 'string'`, it implies the eqivalent of: `LEXICON.term = 'string' and LEXICON.termno = ELEMENTS.termno`; the same is true for the TEXTS table.

Using the information captured in the above tables, we can translate the IR operations described in Section 2.3 into SQL queries. We use simple examples to illustrate. Figure 4(a) is a translation of the CONTAINS operator. It shows how the containment between an element and a text word can be expressed in SQL. Figure 4(b) is a translation of the CONTAINEDIN operator. Figure 4(c) shows the SQL counterpart of the IS operation. Finally, Figure 4(d) shows the DISTANCE operation in SQL. More complex queries involving one or more IR operations can be built up from the simple ones, and the process is very mechanical. Figure 4(e) and (f) express the Antonio and Knuth queries in SQL.

This simple translation of IR queries into SQL allows an IR application to leverage all the advantages that an RDBMS has to offer, including query optimization, and parallel and distributed processing. In addition, since concurrency control and recovery are provided by the RDBMS, updates are much easier. Because traditional IR systems are read-only, systems such as search engines that support high throughput and long-standing service must use large amounts of hardware to keep duplicate copies of indices [RRS98]. Strong support for updates can potentially cut down the requirement for hardware resources and improve the freshness of data.

In Section 4.2 we shall discuss another approach proposed in [FKM99]. The approach separates the postings of each element and each text word in its own table. For example, we would have a LINE table for the "<line>" element, a CLEOPATRA table for the word "cleopatra", etc. We call these tables *term tables*. The schema of these tables are the same as ELEMENTS and TEXTS except there is no need for the `termno` column.

<line> CONTAINS "cleopatra"

```
select line.*
from ELEMENTS line, ELEMENTS cleo
where line.term = 'line'
and   cleo.term = 'cleopatra'
and   line.docno = cleo.docno
and   line.beginno < cleo.wordno
and   cleo.wordno < line.endno
```

(a)

<cite> CONTAINEDIN <article>

```
select cite.*
from  ELEMENTS cite, ELEMENTS article
where cite.term = 'cite'
and   article.term = 'article'
and   cite.docno = article.docno
and   article.beginno < cite.beginno
and   cite.endno < article.beginno
```

(b)

<speaker> IS "antonio"

```
select speaker.*
from ELEMENTS speaker, TEXTS antonio,
where speaker.term = 'speaker'
and   antonio.term = 'antonio'
and   speaker.docno = antonio.docno
and   antonio.wordno - speaker.beginno = 1
and   speaker.endno - antonio.wordno = 1
```

(c)

"knuth" followed by "donald" within 2 words

```
select donald.*, knuth.*
from TEXTS donald, TEXTS knuth,
where donald.term = 'donald'
and   knuth.term = 'knuth'
and   donald.docno = knuth.docno
and   knuth.wordno - donald.wordno <= 2
and   knuth.wordno - donald.wordno > 0
```

(d)

Antonio Query: In the speech spoken by antonio, find the line that contains "merchandise".

```
select   line.*
from     ELEMENTS line, ELEMENTS speaker,
         ELEMENTS speech,
         TEXTS antonio, TEXTS merchandise
where    line.term = 'line' and speaker.term = 'speaker'
   and   speech.term = 'speech'
   and   antonio.term = 'antonio'
   and   merchandise.term = 'merchandise'
// all in the same document
   and   line.docno = merchandise.docno
   and   merchandise.docno = speech.docno
   and   speech.docno = speaker.docno
   and   speaker.docno = antonio.docno
// line CONTAINS merchandise
   and   line.beginno < merchandise.wordno
   and   merchandise.wordno < line.endno
// merchandise CONTAINEDIN speech
   and   speech.beginno < merchandise.wordno
   and   merchandise.wordno < speech.endno
// speech CONTAINS speaker
   and   speech.beginno < speaker.beginno
   and   speaker.endno < speech.endno
// speaker IS "antonio"
   and   antonio.wordno - speaker.beginno = 1
   and   speaker.endno - antonio.wordno = 1
```

(e)

Knuth Query: Find author containing "donald" followed by "knuth" within 2 words.

```
select  au.*
from    ELEMENTS au,
        TEXTS donald, TEXTS knuth
where  au.term = 'author'
   and  donald.term = 'donald'
   and  knuth.term = 'knuth'
// in the same document
   and  au.docno = donald.docno
   and  donald.docno = knuth.docno
// author contains both "donald" and "knuth"
   and  au.beginno < donald.wordno
   and  knuth.wordno < au.endno
// "donald" followed by "knuth" within 2 words
   and  0 <= knuth.wordno - donald.wordno
   and  knuth.wordno - donald.wordno <= 2
```

(f)

Figure 4: Examples of IR operations in SQL (a) CONTAINS (b) CONTAINEDIN (c) IS (d) DISTANCE (e) The Antonio Query (f) The Knuth Query

3.2 Beyond Simple Translation

In this section, we argue for the use of an RDBMS for XML information retrieval on the grounds that it provides capabilities that a conventional IR system cannot offer, and that it is much easier to extend with new IR functionality and features than a custom IR system. We consider three cases.

3.2.1 Case 1. Join Processing

Missing from the operations and languages of most IR systems is the processing of join queries. The join is usually considered a specialty of relational database systems, but is it useful in an IR system as well? We believe the answer is yes. Let's look at an example. Figure 5(a) shows a sample XML document with three bibliography entries. Suppose we want to ask these two queries:

- Find bib entries that cite Smith's paper
- Find authors who have more than one paper in SIGMOD99

These queries require joins and cannot be expressed by conventioned IR systems. The first query implicitly pairs up bib entries in which the <cite> element in one and the <key> element in the other contain the same word. The second query pairs up those bib entries such that the <author> elements in both have the same contents. The challenge for the IR system is that the queries involve searching for keywords that are not constants, but are specified by some conditions. However, these queries can easily be evaluated if we use a relational database system. With the schema and query transformation described in Section 3.1, the two queries can be expressed in SQL as shown in Figure 5(b) and (c), respectively.

3.2.2 Case 2. Queries Mixing Index and Other Data

In the RDB approach, since we store the inverted index in a database system, postings are accessed the same way as regular data. This makes it convenient to access both types of data in the same query, and the database system can automatically take care of optimization. These types of queries are advocated in [GW00] where they are called Web-Supported Database Queries and Database-Supported Web Queries. Again, we use a simple example to illustrate. Suppose we have stored in our database an inverted index of the DBLP web pages, and also a GRADUATES table holding information about graduate students in our department. We can find all graduate students who have a DBLP entry using the SQL query shown in Figure 6.

This query would not be so straightforward if the DBLP postings were indexed in an IR system while the GRADUATES table was kept in a database. In that case, the only natural way would be to build an additional module on top of both systems to pull GRADUATES tuples and the postings out and do the join in this module. In fact, [GW00] is mainly concerned with the construction of this module. This has three disadvantages. First, the module is rather ad-hoc. Second, a join algorithm has to be implemented, and this duplicates something the RDBMS is already good at. Third, it would be costly if there are large number of GRADUATES tuples and postings, or if the database

```
<database>
    <bib>
    <key> John99a </key>
    <title> Semistructured Data </title>
    <author> John </author>
    <publish> SIGMOD99 </publish>
    <cite> Smith74 </cite>
    </bib>
    <bib>
    <key> John99b </key>
    <title> Continuous Optimization </title>
    <author> John </author>
    <publish> SIGMOD99 </publish>
    </bib>
    <bib>
    <key> Smith74 </key>
    <title> Computation Theory </title>
    <author> Smith </author>
    <publish> JACM </publish>
    </bib>
</database>
```

(a)

Find those bib entries that cite Smith's paper.

```
select bib1.*
from  ELEMENTS bib1, ELEMENTS bib2, ELEMENTS au,
      ELEMENTS cite, ELEMENTS key, ELEMENTS smith,
      TEXTS t1, TEXTS t2
where bib1.term = 'bib'  and bib2.term = 'bib'
   and au.term = 'author' and cite.term = 'cite'
   and key.term = 'key' and smith.term = 'smith'
// bib1 contains cite
   and bib1.docno = cite.docno
   and bib1.beginno < cite.beginno
   and cite.endno < bib1.endno
// cite IS t1
   and cite.docno = t1.docno
   and t1.wordno-cite.beginno = 1
   and cite.endno-t1.wordno = 1
// bib2 contains key
   and bib2.docno = key.docno
   and bib2.beginno < key.beginno
   and key.endno < bib2.endno
// key IS t2
   and key.docno = t2.docno
   and t2.wordno-key.beginno=1
   and key.endno-t2.wordno = 1
// bib2 contains author
   and bib2.docno = au.docno
   and bib2.beginno < author.beginno
   and au.endno < bib2.endno
// author IS "smith"
   and au.docno = smith.docno
   and smith.wordno-au.beginno = 1
   and au.endno - smith.wordno = 1
// t1 and t2 are the same thing
   and t1.termno = t2.termno
```

(b)

Find those authors who have more than one paper at SIGMOD99.

```
select t1.*, t2.*
from   ELEMENTS bib1, ELEMENTS bib2,
       ELEMENTS au1, ELEMENTS au2,
       ELEMENTS pub1, ELEMENTS pub2,
       TEXTS sig1, TEXTS sig2,
       TEXTS t1, TEXTS t2
where bib1.term = 'bib'  and  bib2.term = 'bib'
  and  au1.term = 'author' and au2.term = 'author'
  and  pub1.term = 'publish'  and  pub2.term = 'publish'
  and sig1.term = 'sigmod99' and sig2.term = 'sigmod99'
// bib1 contains au1
  and  bib1.docno = au1.docno
  and  bib1.beginno < au1.beginno
  and  au1.endno < bib1.endno
// au1 IS t1
  and au1.docno = t1.docno
  and  t1.wordno-au1.beginno = 1
  and  au1.endno-t1.wordno = 1
// bib1 contains pub1
  and  bib1.docno = pub1.docno
  and  bib1.beginno < pub1.beginno
  and  pub1.endno < bib1.endno
// pub1 IS "sigmod99"
  and  pub1.docno = sig1.docno
  and  sig1.wordno-pub1.beginno = 1
  and  pub1.endno-sig1.wordno = 1
// bib2 contains au2
  and  bib2.docno = au2.docno
  and  bib2.beginno < au2.beginno
  and  au2.endno < bib2.endno
// au2 IS t2
  and  au2.docno = t2.docno
  and  t2.wordno-au2.beginno = 1
  and  au2.endno-t2.wordno = 1
// bib2 contains pub2
  and  bib2.docno = pub2.docno
  and  bib2.beginno < pub2.beginno
  and  pub2.endno < bib2.endno
// pub2 IS "sigmod99"
  and  pub2.docno = sig2.docno
  and  sig2.wordno-pub2.beginno = 1
  and  pub2.endno - sig2.wordno = 1
// t1 and t2 are different occurrences of the same term
  and  t1.termno = t2.termno
  and  t1.wordno <> t2.wordno
```

(c)

Figure 5: (a) Sample XML (b)(c) SQL queries for join processing

```
select  g.*
from    GRADUATES g, ELEMENTS au, TEXTS fn, TEXTS ln,
where   au.term = 'author' and fn.term = g.firstname and ln.term = g.lastname
// author contains firstname
   and  au.docno=fn.docno and au.beginno < fn.wordno and fn.wordno < au.endno
// author contains lastname
   and  au.docno=ln.docno and au.beginno < ln.wordno and ln.wordno < au.endno
```

Figure 6: Query joining the GRADUATES table with postings

and the IR system run in different processes or on different platforms. It is much more efficient to push as much processing down into a single system as possible.

3.2.3 Case 3. Direct Containment

Another advantage of using an RDBMS instead of a special purpose IR system is that there is greater extensibility.

One can observe from Section 2 that the containment operators do not distinguish the nesting depth. That is, if posting P1 nests posting P2, P2 may be nested one or several levels below P1. This is fine if this is what is desired, but is a problem if it is not.

Suppose we have XML documents as shown in Figure 7(a), and we want to get the title of the first section. The query "<title> CONTAINEDIN <section>[1]" does not give us the exact answer, but rather a superset, as titles of subsections would also be returned. What we want is something like "<title> DIRECT CONTAINEDIN <section>[1]". Further, notice that if we express the query as "<section>[1] DIRECT CONTAINS <title>", it is exactly "<section>[1].<title>", which is just a path expression. Thus supporting direct containment is important.

Adding direct containment to an IR system that does not provide such support is not trivial. The parser, index structure, storage and operators all need to be extended and partially rewritten. Although the parser needs to be modified no matter what, other changes are easier to deal with if we build the functionality on top of a relational database system.

One solution is to add another column, `depth`, to the ELEMENTS and TEXTS tables. Each posting now has an additional attribute, which indicates the depth of an occurrence of a term in a document. The root element of a document has depth 0, the other elements and text words have depths relative to the root. For the sample XML in Figure 7(a), the first <section> has a depth of 0 since it is the root. The <title> containing "Information Retrieval on RDB" has depth 1, as well as the second <section>. The text word "Case" has depth 4. The query "<title> DIRECT CONTAINEDIN <section>[1]" can be expressed in SQL as shown in Figure 7(b). It is straightforward how to extend the operation CONTAIN_FUNC (Section 2.3) to handle direct containment and therefore we omit it.

```
<section>
  <title> Information Retrieval on RDB </title>
  <section>
    <title> Beyond Simple Translation </title>
    <section>
      <title> Case 1. Join Processing </title>
    </section>
  </section>
</section>
```

(a)

```
select  title.*
from    ELEMENTS section, ELEMENTS title
where  section.term = 'section'
   and   title.term = 'title'
   and   section.beginno = (select min(beginno)
                            from   ELEMENTS e
                            where e.term = 'section')
   and   section.docno = title.docno
   and   section.beginno < title.beginno
   and   title.endno < section.endno
   and   title.depth - section.depth = 1
```

(b)

Figure 7: (a) A sample XML document (b) SQL query for `<title> DIRECT_CONTAINEDIN <section>[1]`

4 Performance Evaluation of Information Retrieval Using RDB

In this section, we describe our performance study that aims to compare the retrieval performance using a database system versus a special purpose IR system. We used two commercial database systems for the study. However, due to space constraints, we only report the results using one of them, DB2 UDB version 6.1. Experiment results on the other commercial DBMS were similar. We ran DB2 UDB on a 500MHZ PIII machine with 256MB memory and Linux Redhat 6.1.

Since we could not find an appropriate commercial-strength IR system that supports XML information retrieval, we built our own system, which supports boolean, containment, and proximity queries, but does not yet support ranking. Two lexicons are kept in memory for elements and text words. Two GDBM files are used to store inverted lists, one for element lists and the other for text lists. Each inverted list is stored as a contiguous record. GDBM [GNU] is Gnu's replacement of the dbm and ndbm libraries, which are used in IR systems such as WAIS[1] [PFH95]. GDBM provides a set of database functions using extendible hashing and allows arbitrary sized keys and data items. It allows multiple readers to access data stored in a GDBM file, but a writer has exclusive access. We assume that each inverted list fits in memory, and is read in as a whole when retrieved. The IR system runs on the same machine as DB2 UDB, and it is a single server system, that is, there is no client. The GDBM block size is set at 8 KB, and the cache size is set to zero to eliminate additional caching above the OS file system.

Both the IR and the database systems share the same parsing process, which takes the XML data set and produces either the IR inverted index stored in GDBM, or load files for the RDBMSs. Case folding and stop words filtering are done when processing the XML datasets. The postings in the load files are in the same sequence as the terms are parsed, thus they are ordered according to the `docno` column. After the postings are loaded into the database, clustered indices are built. The postings are then rearranged in another order indictated by the index key.

A large number of factors affect performance in an RDBMS. Index type, buffer pool size, heap

[1] http://ls6-www.cs.uni-dortmund.de/ir/projects/freeWAIS-sf/index.html

	Shakespeare	DBLP	Synthetic
Total size	8 MB	53 MB	207 MB
Number of XML files	37	568	500
Number of distinct elements	22	598	1001
Number of distinct, nonstopped text words	22,825	250,657	500,000
Total number of elements	179,726	1,595,010	2,200,298
Total number of nonstopped text words	474,057	3,655,148	19,699,413

Table 1: Datasets

size, transaction type, isolation level, optimization level, and the way the query is written could all affect performance. We chose to use the default settings for most parameters in the RDBMSs, except that the buffer pool size is 16 MB and that the hash join is enabled. We found that variations on the buffer pool size and sort heap size do affect results (we experimented with these sizes varied up to 100 MB), but the impact is insignificant and increasing the sizes could adversely affect the results. Slight improvements can be observed for some queries, and degradation can be observed for others.

4.1 Data Sets and Queries

We used three XML datasets in our study. The first is a set of Shakepeare plays[2]. The second is a set of DBLP bibliographies[3]. We also generated some synthetic XML documents since we could not find a larger and complete set of XML data. Table 1 lists the sizes of the datasets. Note that the total number of elements and text words are exactly the cardinalities of the ELEMENTS and TEXTS tables.

The synthetic data generator first produces a random tree in which each node corresponds to an element. It then uses this tree as a template and varies the occurrences of leaf nodes (the frequency of leaf elements) and the text content of nodes to generate each document. The template tree has 7 levels. The following two constraints are also made to the synthetic dataset:

- All text words in the whole set of data follow a zipfian distribution. We used the generalized formulation of Zipf's law with the constant 1.0 [Poo].
- Three elements are controlled to appear 20, 2000 and 200000 times in the set.

We chose to use simple queries for our performance study, as they allowed us to see the behavior of different systems more clearly than complex ones. We focused on fourteen containment queries of varying input/output sizes. However, we also tested some complex queries. Results from two of them, the Antonio query and the Knuth query, are shown in Section 4.3.

Each of the fourteen queries is coded "QXN" where 'X' is one of 'S'(Shakespeare), 'D'(DBLP), or 'G' (generated data), and 'N' is the query number within the respective dataset. All fourteen queries

[2]http://www.oasis-open.org/cover/xml.html.

[3]ftp://ftp.informatik.uni-trier.de/. The dataset is a modified DBLP archive. The original archive consists of 141,023 small files averaging 374 bytes each. We combined small files into bigger ones to obtain a more realistic average of 93KB per document.

	term1 frequency	term2 frequency	num results: term1 CONTAINS term2
QS1	1	147	1
QS2	107,833	277	36
QS3	107,833	3,231	1,543
QS4	107,833	1	1
QD1	5	18	1
QD2	4,188	712	672
QD3	287,513	6,363	6,315
QD4	287,513	3	3
QG1	20	13	1
QG2	2,000	10,068	7
QG3	200,000	104,278	10,419
QG4	200,000	1,402,124	137,970
QG5	200,000	13	1
QG6	20	1,402,124	0

Table 2: Query Sizes

are of the form "term1 CONTAINS term2" where "term1" is an element and "term2" is a text word. The translated SQL query looks like that shown in Figure 4(a). Table 2 shows the frequencies of the terms and the number of results produced by each query. Dividing the frequency of each term by the total number of elements or text words in Table 1, we get the *selectivity* of each term, by which we mean the ratio of the frequency of a term to the cardinality of the table that holds this term. The greater the ratio, the less selective the term.

Except for QS4, QD4, QG5 and QG6 (the last one or two of each query set), queries within a set operate on increasingly larger inputs and produce increasingly larger outputs. Queries QS4, QD4, QG5 and QG6 stand out from the others. In these queries, one term is highly selective, while the other is highly unselective.

We tried different indices on the ELEMENTS and TEXTS tables but we consider two representative ones in this paper. One is a clustered index on the `(termno, docno)` columns, the other is a clustered index on all columns. We call the former *two-column(2col) index* and the latter *cover index*. Table 3 shows the storage requirements for the inverted files in the IR system and for the tables and indices in DB2. No compression is done on the inverted lists. As we can see, the cover index requires much more storage than the two-column index, in fact, it is at least as big as the table itself. Also, the RDBMS requires much more storage than the IR system, especially with the index.

Note that the term "index" is overloaded in this paper. There is the IR inverted index, then there are indices in the RDBMS. To make things clear, we use qualifications as much as possible. In cases where there is no qualification, the term refers to the RDBMS index.

	IR	DB2		
	inverted file	table only	2col index	cover index
Shakespeare Elements	1	5	1	5
Shakespeare Texts	4	10	4	10
DBLP Elements	18	41	8	42
DBLP Texts	42	78	30	79
Synthetic Elements	132	56	18	57
Synthetic Texts	335	421	178	428

Table 3: Storage [MB]

	lists retrieval(hot)	lists retrieval(cold)	in-mem operation	total(hot)	total(cold)
QS1	0.3	30.2	0.03	0.3	30.3
QS2	47.8	112.6	34.3	82.1	146.9
QS3	46.3	120.1	361.7	408.0	478.8
QS4	46.0	120.8	0.6	46.6	121.4
QD1	0.3	36.0	0.02	0.3	36.0
QD2	1.8	59.0	46.8	48.7	105.8
QD3	133.2	312.3	522.9	656.1	835.3
QD4	129.9	297.8	0.7	130.7	298.6
QG1	0.3	108.2	0.03	0.3	108.2
QG2	5.7	178.6	2.6	8.3	181.2
QG3	110.8	322.8	1285.3	1396.1	1608.1
QG4	450.1	929.6	22207.7	22657.8	23137.3
QG5	81.2	274.8	0.7	81.9	275.5
QG6	380.0	750.0	0.1	380.0	750.0

Table 4: IR timings [msec]

4.2 Base Query Performance

Table 4 shows the performance of the IR system for the fourteen queries. Two timings are shown for inverted lists retrieval, the "hot" times are measured when the queries are run multiple times. The "cold" times are measured when the queries are run after the memory is flushed. The difference between the hot and cold retrieval times reflects the effect of OS file system caching. The column "in-mem operation" shows the time it takes to do the IR containment operation, after the inverted lists are retrieved from disk. The "total"s are the sum of in-memory operation and list retrieval times.

Table 5 show the results of running the fourteen queries on DB2, as well as the ratios between DB2(hot) timings and IR(hot) timings. The "IR" column repeat the "total(hot)" times from Table 4. The timings are total elapsed times, which include query execution and result fetching times. Since the production of result tuples can go on in parallel with the fetching of result tuples, the total elapsed times must be measured.

A few important points can be observed from these numbers. First, for queries with more selective

	IR	DB2(no index)			DB2(2col)			DB2(cover)		
	hot	cold	hot	ratio	cold	hot	ratio	cold	hot	ratio
QS1	0.3	1828	742	2506.8	285	4	13.5	202	1	3.4
QS2	82	2011	1454	17.7	3010	2584	31.5	636	414	5.0
QS3	408	7275	6788	16.6	12099	10501	25.7	11350	11040	27.1
QS4	47	2202	874	18.6	2733	1965	41.8	203	3	0.1
QD1	0.3	8895	6462	21256.6	483	2	6.6	244	1	3.3
QD2	49	9661	6796	138.7	1985	892	18.2	1143	782	16.0
QD3	656	17649	15482	23.6	242985	237378	361.9	6965	6259	9.5
QD4	131	9611	6830	52.1	741	67	0.5	994	2	0
QG1	0.3	38365	36896	110467.1	1042	2	6.0	267	5	16.6
QG2	8	38932	37273	4659.1	16394	1455	181.9	1035	675	84.4
QG3	1396	78176	75773	54.3	743471	670530	480.3	13206	12056	8.6
QG4	22658	555977	552300	24.4	563868	550542	24.3	623271	623026	27.5
QG5	82	39831	36940	450.5	1383	72	0.9	927	2	0
QG6	380	39491	36259	95.4	39874	31113	81.9	299	2	0

Table 5: DB2 performance. Cold and hot times are in msec. Ratios are results of DB2(hot) times divided by IR(hot) times.

terms, using the RDBMS index is definitely a good idea. However, for queries with unselective terms, it may be better off scanning the whole table (compare the numbers using no index with those using an index).

Second, for DB2, the queries perform significantly better using the cover index than using the two-column index. We believe this is because with DB2, when the cover index is used, all values can be found in the index. Whereas when the two-column index is used, tuples need to be fetched from the tables, and this cost is additional to the cost of scanning the two-column index.

Third, DB2 performs significantly better for queries QS4, QD4, QG5 and QG6 than the IR system. The key point to note is that in the IR system, there is no index on the postings (extendible hashing is used to locate the inverted lists), while in the databases, postings are stored in the relational tables, on which indices are built. As described in Section 2.3, the IR system uses an algorithm analogous to the merge phase of a sort-merge join to process containment, thus all postings are retrieved and operated upon. DB2, on the other hand, discerns the difference in the selectivities of the two terms, and uses a nested loop join on ELEMENTS and TEXTS tables, arranging the table containing the more selective term as the outer. In addition, the index nested loop join allows all of the predicates in the query to be either the sargable or the start- and stop-key predicate on the inner. Since the outer term is highly selective, and the predicates have high filtering factors, the index nested loop join enables only a small number of tuples to be fetched from the inner table, thus avoiding a large amount of I/O and CPU cost.

Note that it is not true that the RDBMS beats IR whenever index nested loop join is used on a query involving terms of different selectivities. The savings from fetching/operating on only a portion of the postings could lose to other overhead in the RDBMS. Obviously, there is a crossover

	no index		index on docno		index on all	
	cold	hot	cold	hot	cold	hot
QS1	58	1	58	1	35	1
QS2	2069	1794	2117	1784	2836	2351
QS3	8700	8382	6424	6180	11149	10970
QS4	1126	219	260	48	177	3
QD1	53	1	51	1	43	1
QD2	1084	960	1217	1053	1353	1240
QD3	14072	13813	27872	27336	18654	18502
QD4	1269	646	401	82	192	3
QG1	52	1	51	1	52	1
QG2	319	151	376	151	393	180
QG3	43293	42778	43508	43130	60581	60209
QG4	518264	515189	553909	550458	630421	629590
QG5	726	452	394	96	510	2
QG6	3798	2759	172	4	124	2

Table 6: Performance using term tables [msec]

point.

Fourth, consider queries other than QS4, QD4, QG5 and QG6. Using either the two-column or the cover index, DB2 timings for 6 out of 12 of them are within 10 times those of IR. We think that this is encouraging. However, Table 5 also shows that the gap could be big for queries that the DBMS does not perform well.

Recall from Section 3.1 that term tables are ones that contain postings for only one term. Having one table per term is generally not practical for the RDBMS as there would be too many of them. The term tables approach is proposed in [FKM99], and is similar to the "Binary" approach in [FK99]. Since the term tables approach is reported to be effective in [FKM99] and the Binary approach is shown to be the best in [FK99], we tried with our own experiments. Another motivation for trying the term tables approach is that, if this approach performs well in the RDBMS, a hybrid alternative could be used in which postings for frequently accessed terms are stored in separate tables, while the ELEMENTS and TEXTS tables still keep all the postings.

Table 6 shows the results of this experiment. We can see that the performance for the term tables approach is better than using the two-column index on the big tables, but is worse than using the cover index. Given this, and the problems of storing a large number of tables, we see no clear advantage of the term tables approach.

4.2.1 Discussion

During our performance study, we observed that, although very different query plans could be generated for the same query with different indices, and with the same index but different join methods, the actual running times are not always very different. We also observed that the optimizers in the two commercial DBMSs make very different decisions. We want to point out that the distribution of term frequencies is highly skewed in the ELEMENTS and TEXTS tables, and we observed that the optimizer estimations are often far from being accurate.

DB2 reported CPU costs over 80% of elapsed times for over 60% of the queries, especially the large ones. This does not mean that I/O cost is the remaining 20%, because the RDBMS can overlap CPU and I/O. This could nevertheless indicate that those database system features costing CPU time, such as locking and latching, interpretive predicate evaluation, and result binding, do add a significant amount of overhead.

Due to space constraints, we did not include the two component timings, execution time and result-fetch time, in Tables 5, 6, and 7. However, we remark that result-fetch times are the dominant cost for queries with large output sizes (e.g., QG3 and QG4). This is worrisome because web-oriented queries could incur large number of results. Delay in exporting the results is a significant obstacle to utilizing an RDBMS for information retrieval.

From the case where DB2 is faster than IR, one may argue that if a higher level index is added on top of IR postings, the IR system would also perform well. However, trade-offs must be considered. An additional index adds storage overhead, further, the storage and retrieval of the inverted lists are also likely to be affected. Perhaps by adding database-type indices, storage in IR system would get closer to that in the RDBMS. So why don't we improve RDBMS to better support the IR workloads?

We think that query processing in the RDBMS could be improved. Notice that the IR System stores the inverted lists in sorted order (Section 2.3). This arrangement is critical to ensuring that it only does the analogue of the merge phase of a sort-merge join for the IR operations, and that the cost of the operations is linear in the number of postings. Neither of the DBMSs we tried used this algorithm. This is probably because they do not realize that, when we select on a single text word or element, the postings are already stored in the desired order. To improve, a DBMS could use the index to seek to the beginnings of two tuple "list"s in question, and do a merge on them, applying the equi join predicate on `docno` and the non-equi join predicates on `beginno, endno, wordno` while merging. The cost of sorting and filtering *before* the merge join could be avoided. The indices are not used to fetch tuples from the tables, but rather are used to identify the start and stop points of table scans.

In summary, we think that the following areas are worthwhile to be improved for the DBMS: reduction of CPU costs, better statistic support for highly skewed data, faster path for exporting large answer sets, and IR-specific processing algorithms and optimizations. The impact and actual benefit of these need to be evaluated and measured on a system with carefully designed experiments. In addition, it is worthwhile to investigate whether using multi-dimensional indices on the `begin, end`, and `wordno` columns can help the containment queries. These issues are avenues for future work.

	IR	DB2, 2col index	DB2, cover index
Antonio (Shakespeare)	73	4546	1631
Knuth (DBLP)	135	3213	96

Table 7: Performance of the Antonio query and the Knuth query [msec]. All numbers are hot times.

4.3 Performance for More Complex Queries

Besides the fourteen simple containment queries, we also experimented with more complex queries. We report two in this section. The SQL counterparts of the Antonio and Knuth queries are shown in Figure 4(e) and (f), respectively, and the timings are listed in Table 7. Notice that when the cover index is used, DB2 performs better than IR.

From the results of these two and other complex queries, we observed that the running times do not depend on the complexity of the queries. Queries that look simple may take a long time to execute, while queries that look complex may run very quickly. What is more important is the selectivity of the terms and the sizes of inputs and outputs.

The results from complex queries are encouraging because more effective retrieval can result from these queries without necessarily incurring poor query evaluation performance.

5 Conclusion

We have shown that there are a number of advantages for using a database system for XML information retrieval. In addition, there are advantages we can gain from the RDBMS that are not offered by traditional IR systems. However, our performance study shows that in general the database systems are not well tuned for IR queries. The performance of the RDBMSs is close to or better than that of the IR system for queries involving selective terms, but the gap could also be big for queries on which the RDBMSs do not perform well. The cases where RDBMSs outperform the IR system demonstrate the importance of query optimization and indices. We believe that with additional research into techniques for supporting IR workloads in relational systems, RDBMSs could become a viable alternative to special purpose inverted list systems for supporting XML information retrieval.

References

[AFL+94] T.Arnold-Moore, M.Fuller, B.Lowe, J.Thom, R.Wilkinson, *The ELF Data Model and SGQL Query Language for Structured Document Databases*, CITRI TR 94-13, Collaborative Information Technology Research Institute, Melbourne, Australia, 1994.

[BCK+94] G. E. Blake, M. P. Consens, P. Kilpelainen, P.-A. Larson, T. Snider, F. W. Tompa, *Text/Relational Database Management Systems: Harmonizing SQL and SGML*, Proceedings of the International Conference on Applications of Databases, pp.267-280, Vadstena, Sweden, June, 1994.

[BN96] Ricardo Baeza-Yates, Gonzalo Navarro, *Integrating Contents and Structure in Text Retrieval*, SIGMOD Record, Vol. 25, No.1, March 1996.

[CCB95a] C. L. Clarke, G. V. Cormack, F. J. Burkowski, *An Algebra for Structured Text Search and A Framework for Its Implementation*, The Computer Journal, 38(1):43-56, 1995.

[CCB95b] C.L.Clarke, G.V.Cormack, F.J.Burkowski, *Schema-independent retrieval from heterogeneous structured text*, Fourth Annual Symposium on Document Analysis and Information Retrieval, pages 279-289, 1995.

[D98] Tuong Dao, *An Indexing Model for Structured Documents to Support Queries on Content, Structure and Attributes*, Proceedings of ADL'98, April 1998, Santa Barbara, California.

[DM97] Stefan Dessloch, Nelson Mattos, *Integrating SQL Databases with Content-specific Search Engines*, Proceedings of the 23rd VLDB Conference, Athens, Greece,1997.

[DST96] Tuong Dao, Ron Sacks-Davis, James A. Thom, *Indexing Structured Text for Queries on Containment Relationships*, Proceedings of the 7th Australasian Database Conference, Melbourne, Australia, 1996.

[FK99] Daniela Florescu, Donald Kossman, *Storing and Querying XML Data Using an RDBMS*, IEEE Data Engineering Bulletin 22(3):27-34(1999).

[FKM99] Daniela Florescu, Donald Kossmann, Ioana Manolescu, *Integrating Keyword Search into XML Query Processing*, INRIA Technical Report, INRIA, December, 1999.

[GNU] GDBM, http://www.gnu.org/software/gdbm/gdbm.html.

[GW00] Roy Goldman, Jennifer Widom, *WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web*, to appear in Proceedings of the 2000 Sigmod Conference.

[K73] D.E.Knuth, *The Art of Computer Programming. Volume III: Sorting and Searching*, Addison-Wesley, 1973.

[M90] I.A.Macleod, *Storage and Retrieval of Structured Documents*, Information Processing & Management 26(2):197-208, 1990.

[MAG+97] J.McHugh, S.Abiteboul, R.Goldman, D.Quass, J.Widom, *Lore: A Database Management System for Semistructured Data*, SIGMOD Record, 26(3):54-66.

[MWA+98] J.McHugh, J.Widom, S.Abiteboul, Q.Luo, and A.Rajaraman, *Indexing Semistructured Data*, Stanford Technical Report, January 1998.

[PFH95] Ulrich Pfeifer, Norbert Fuhr, Tung Huynh, *Searching Structured Documents with the Enhanced Retrieval Functionality of freeWAIS-sf and SFgate*, The Third International World Wide Web Conference, Technology, Tools and Applications, Darmstadt, Germany April 10-14, 1995.

[Poo] Viswanath Poosala, *Zipf's Law*, Technical Report.

[RRS98] Eric J. Ray, Deborah S. Ray, Richard Seltzer, *AltaVista Search Revolution,Second Edition*, Osborne McGraw-Hill, April 1998.

[SAZ95] Ron Sacks-Davis, Timothy Arnold-Moore, Justin Zobel, *Database Systems for Structured Documents*, Proceedings of the International Symposium on Advanced Database Technologies and Their Integration (ADTI'94), pp. 272-283, Nara, Japan, October 1994.

[SGML86] International Organization for Standardization, *Information processing–text and office systems–Standard Generalised Markup Language (SGML)*, ISO/IEC 8879: 1986.

[SGT+99] Jayavel Shanmugasundaram, He Gang, Kristin Tufte, Chun Zhang, David DeWitt, Jeffrey Naughton, *Relational Databases for Querying XML Documents: Limitations and Opportunities*, Proceedings of the 1999 VLDB Conference, September 1999.

[SM83] G. Salton, M.J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, 1983.

[TDC00] Feng Tian, David DeWitt, Jianjun Chen, Chun Zhang, *The Design and Performance Evaluation of Various XML Storage Strategies*, submitted for publication.

[XML98] T.Bray, J. Paoli, C.M.Sperberg-McQueen, *Extensible Markup Language (XML) 1.0*, http://www.w3.org/TR/REC-xml.

[YA94] Tak W. Yan, Jurgen Annevelink, *Integrating a Structured-Text Retrieval System with an Object-Oriented Database System*, VLDB'94. Santiago, Chile, September, 1994.