On Supporting Containment Queries in Relational Database Management Systems

Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo
Department of Computer Sciences
University of Wisconsin, Madison
czhang,naughton,dewitt,qiongluo@cs.wisc.edu

Guy Lohman
IBM Almaden Research Center
San Jose, California
lohman@us.ibm.com

ABSTRACT

Virtually all proposals for querying XML include a class of query we term "containment queries". It is also clear that in the foreseeable future, a substantial amount of XML data will be stored in relational database systems. This raises the question of how to support these containment queries. The inverted list technology that underlies much of Information Retrieval is well-suited to these queries, but should we implement this technology (a) in a separate loosely-coupled IR engine, or (b) using the native tables and query execution machinery of the RDBMS? With option (b), more than twenty years of work on RDBMS query optimization, query execution, scalability, and concurrency control and recovery immediately extend to the queries and structures that implement these new operations. But all this will be irrelevant if the performance of option (b) lags that of (a) by too much. In this paper, we explore some performance implications of both options using native implementations in two commercial relational database systems and in a special purpose inverted list engine. Our performance study shows that while RDBMSs are generally poorly suited for such queries, under certain conditions they can outperform an inverted list engine. Our analysis further identifies two significant causes that differentiate the performance of the IR and RDBMS implementations: the join algorithms employed and the hardware cache utilization. Our results suggest that contrary to most expectations, with some modifications, a native implementation in an RDBMS can support this class of query much more efficiently.

1. INTRODUCTION

In query languages proposed for XML, and even more generic SGML query languages, containment queries play a prominent role. By "containment query" we mean queries that are based on the containment and proximity relationships among elements, attributes, and their contents.

While there is a great deal of work being done on how to support such query languages in special purpose query engines [14, 17, 27, 20, 22, 21], it is clear that in the foreseeable future a great deal of XML data will be stored in relational database systems. (As evidence for this, notice that major DBMS vendors including IBM, Microsoft, and Oracle are investing substantial resources toward improving their system support for XML data.)

Since containment queries are an important aspect of querying XML data, and RDBMSs must support the storage and querying of XML data, the question arises: how should we support containment queries in RDBMS? The inverted list technology that underlies much of Information Retrieval is well-suited to containment queries, but should we implement this technology (a) in a loosely-coupled IR engine, or (b) using the native tables and query machinery of the RDBMS?

Commercial products such as DB2 Text Extender and SQL Server Full-Text Search Service take option (a), where a special purpose IR engine is coupled with the database engine. Users employ certain operators (e.g.,"contains", "formsof", "synonym form of") to tell the system to route parts of the query to the IR engine. Query results from both engines are then combined before returning to the user. In this type of system, a "glue" module must be used between the IR engine and the main database engine to handle locking, concurrency control and recovery. In addition, the optimizer has no choice as to which engine should execute which part of a query, since the use of the "IR" engine is dictated by the use of special predicates when the query was framed.

Option (b) directly utilizes the database storage and processing power to process containment queries. More than twenty years of work on query optimization, query execution, scalability, and concurrency control and recovery immediately extend to the queries and structures that implement the new operations. In addition, this approach has the advantage of one unified optimization and execution framework for all queries. However, all this will be irrelevant if the performance of the native implementation lags that of the "coupled" implementation by too much.

In this paper we take a first step in exploring the two options. We compare the implementation of containment queries using native support in two commercial relational database systems and in a special purpose inverted list engine. We transform the inverted index into relational tables and convert containment queries into SQL queries. It turns out that a native implementation using an RDBMS can process queries that are either difficult or impossible to process by an inverted list engine.

Our performance study shows that while RDBMSs are generally poorly suited for containment queries, under cer-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA
Copyright 2001 ACM 1-58113-332-4/01/05 ...$5.00.

tain conditions they can outperform the inverted list engine. Also, our experiments point to two significant causes that differentiate the performance of the IR and RDBMS implementations: (1) the join algorithms employed, and (2) the hardware cache utilization achieved.

The inverted list engine uses a merge join that we term *Multi-Predicate Merge Join* (MPMGJN), as its workhorse join operator. This algorithm is different from the standard merge join and the index nested-loop join algorithms, and the difference has a significant impact on performance. To isolate algorithmic differences from other factors, we implemented the standard merge join and the index nested-loop join ourselves. We compare the join algorithms in great detail and our experimental results show that MPMGJN could out-perform the standard RDBMS join algorithms by more than an order of magnitude on containment queries.

Additionally, with main memory sizes getting larger and the memory hierarchy getting deeper, many researchers have recognized the importance of hardware cache utilization on performance [2, 31, 25]. We conducted experiments to see whether there is a significant difference between the inverted list engine and the RDBMS. The answer is yes. To the best of our knowledge, this is the first paper that studies and compares a merge join using multiple predicates with standard join algorithms and shows their impact on hardware cache utilization.

The rest of the paper is organized as follows: Section 2 describes containment queries, their processing using the inverted index, and the translation of the inverted index into relations and containment queries into SQL queries. Section 3 details our performance study and analysis comparing the two types of systems: an inverted list engine and the RDBMS. Section 4 discusses related work, and Section 5 concludes.

2. CONTAINMENT QUERY PROCESSING

Containment queries are a class of queries based on containment relationships among elements, attributes, and their contents. This class subsumes database path expressions (including regular path expressions) and Information Retrieval Boolean and proximity queries. In this section we first use examples to illustrate these queries and establish their importance, we then show how to use the inverted index and the relational database system to process basic containment queries. Since complex containment queries can be reduced to a sequence of more basic ones, and basic containment queries shed more light on the fundamental determinants of performance, we mostly focus on simple containment queries such as "title *contains* 'galaxy' ".

2.1 Containment Queries

Descriptions of path expressions can be found in Lorel [1] and XQuery [5]. Since XQuery is a query language proposed by W3C, the XQuery style of path expressions is likely to be widely used. Thus we adopt its syntax to describe the containment queries.

The following is a containment query:

```
Q1.  /doc[author='John Smith']//section/title
```

It selects titles of sections contained in the document authored by "John Smith". In this query, the leading "/" indicates that "doc" must be a root element. "[author='John Smith']" is a predicate restricting "doc" elements to those that contain "author" elements, whose content is "John Smith". The symbols "/" and "//" represent containment, with "/" indicating direct containment (i.e., parent-child relationship), and "//" indicating indirect containment (i.e., predecessor-descendent relationship).

The following is an IR Boolean and proximity query:

```
Q2.  /doc[author/('Smith' NOT 'Adams') AND
distance('UNIX', 'DOS')≤5 ]
```

It selects all documents whose authors include "Smith" but not "Adams", and this document must also have the two words "UNIX" and "DOS" within a distance of five words.

LoreL [1] defines a class of useful regular path expressions, which can use wildcards '?', '+', and '*' to mean repetitions of zero or one, one or more, and zero or more, respectively. For example the expression

```
Q3.  chapter[1]/section*/title
```

selects the titles of the first chapter, as well as the titles of sections and subsections in the chapter. Note that this expression differs from `chapter[1]/section/*/title`, which selects the titles of anything contained in a section of the first chapter, including the titles of captions or figures. Note that since there is no leading "/", "chapter" can be anywhere inside a document.

At the heart of these queries is a simple containment query: does one specified element or word appear within another specified element?

2.2 Containment Query Processing Using the Inverted Index

The inverted index [28] is very popular in information retrieval systems as it supports Boolean, proximity, and ranking queries efficiently. The classic inverted index data structure maps a text word (or a phrase) to a list, which enumerates documents containing the word and its position within each document.

In order to process structured documents such as XML, the inverted index can be extended in a simple way: text words are indexed in a T-index similar to that used in a traditional IR system, and elements are indexed in an E-index, which maps *elements* to inverted lists. Figure 1 illustrates the structure of the two indexes for a sample XML file.

Each inverted list records the occurrences of a word or an element—here we use "term" to refer to both of them. Each occurrence is indexed by its document number, its position and its nesting depth within the document. This is denoted in Figure 1 as $(docno, begin : end, level)$ for an element and $(docno, wordno, level)$ for a text word. The position, $begin : end$ or $wordno$, in a document can be generated by counting word numbers. Alternatively, if the document is in a parsed tree format, the position can be generated by doing a depth-first traversal of the tree and sequentially assigning a number at each visit. Since each non-leaf node is always traversed twice, once before visiting all its children and once after, it has two numbers assigned, while leaf nodes have only one number. An inverted list is sorted in the increasing order of $docno$, and then in the increasing order of $begin$ and end[1].

Term occurrences indexed in this way have the following properties:

1. **Containment Property**. An occurrence of a term T_1, encoded as (D_1, P_1, L_1), *contains* an occurrence of a term T_2, encoded as (D_2, P_2, L_2), if and only if: (1)

[1] Since XML documents are strictly nested, sorting in the order of $(docno, begin)$ suffices.

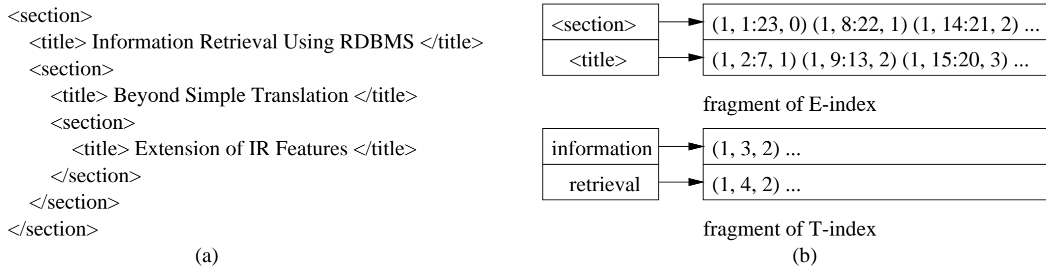

Figure 1: (a) A sample XML document, (b) its inverted lists in T-index and E-index.

$D_1 = D_2$, and (2) P_1 nests P_2. For example, $(1, 1 : 23, 0)$ contains $(1, 9 : 13, 2)$.

2. **Direct Containment Property**. An occurrence of a term T_1 (D_1, P_1, L_1) *direct_contains* T_2 (D_2, P_2, L_2) if and only if: (1) $D_1 = D_2$, (2) P_1 nests P_2, and (3) $L_1 + 1 = L_2$. For example, $(1, 1 : 23, 0)$ direct_contains $(1, 2 : 7, 1)$.

3. **Tight Containment Property**. An occurrence of a term T_1 (D_1, P_1, L_1) *tight_contains* T_2 (D_2, P_2, L_2) if and only if: (1) $D_1 = D_2$, and (2) P_1 nests P_2 and nothing else. For example, $(1, 14 : 21, 2)$ tight_contains $(1, 15 : 20, 3)$. Because of the nesting structure of XML, tight containment implies direct containment but not vice versa.

4. **Proximity Property**. An occurrence of a term T_1 (D_1, P_1, L_1), is within distance k of a term T_2 (D_2, P_2, L_2), if and only if: (1) $D_1 = D_2$, and (2) $|P_1 - P_2| \leq k$. For example, $(1, 3, 2)$ and $(1, 4, 2)$ are within distance of 1 (appear next to each other). The exact definition of proximity on elements and attributes depend on the application supported.

The above properties allow us to have a variety of operations on inverted lists. To process the expression "a//b", the inverted lists of "a" and "b" are retrieved. Occurrences from the two lists are merged if they satisfy the **Containment Property**. The expression "a/b" can be similarly processed by merging the inverted lists using the **Direct Containment Property**. The **Proximity Property** can be used to process string queries such as "query processing" with distance $k = 1$. Finally the **Tight Containment Property** can be used to process expressions such as " <month>='january' " (element '<month>' has only 'january' in it and nothing else).

A framework using inverted lists to process containment queries can be constructed using these operations as basic building blocks, and we have implemented a prototype system. Central to the framework is the merging of two inverted lists. If we view an inverted list as an ordered relation, the merging of two inverted lists is effectively a join and the properties used for merging are essentially join predicates. Since this type of join is used extensively, its efficiency has paramount importance. In Section 3, we compare the performance of this type of join in a special-purpose inverted list engine and in two commercial RDBMSs.

2.3 Containment Query Processing Using an RDBMS

In this section, we first introduce the relational schema used to store the inverted index and the mapping from containment queries to SQL queries, we then illustrate two additional types of queries that can be processed as a result of utilizing a powerful and extensible RDBMS.

2.3.1 *Schema and Processing of Containment Queries in the RDBMS*

The E-index and T-index can be mapped into the following two relations:

ELEMENTS `(term, docno, begin, end, level)`
TEXTS `(term, docno, wordno, level)`

The `ELEMENTS` table stores occurrences of XML elements, while the `TEXTS` table stores occurrences of text words. Each occurrence is stored as a table row.

Figures 2(a)-(d) show the translations of inverted lists merging operations into SQL. A merge of two inverted lists is translated into a join, and the property applied when merging is translated into join predicates. Here we only show the containment between an element and a text word. It involves joins between the `ELEMENTS` table and the `TEXTS` table. Containment between elements is the same except involving a self-join on the `ELEMENTS` table.

2.3.2 *Leveraging the Power of the RDBMS*

By implementing containment queries using an RDBMS, we are able to process queries that are difficult or impossible to process using only an inverted list engine. We show two examples.

Type 1. Joined Searching.
By joined searching, we mean a query that binds multiple path expressions by a common variable. Such a query can be expressed in languages such as XML-QL and XQuery, but is difficult or impossible to express as a containment query, because it involves searching for terms that are not constants, but are specified by other search conditions. However it is not difficult to express in SQL.

For example, suppose we want to ask the query: "Find those bib entries that cite Smith's paper". This query implies two path expressions: "bib[author/'Smith']/key" and "bib/cite", with the additional requirement that the "key" element have the same content as the "cite" element. The SQL expression of this query can be found in the full version of this paper [36].

Type 2. Queries on Mixed Data.
Since we now store the inverted index in a database system, the index is accessed the same way as other relational data. This makes it convenient to use the inverted index to aug-

```
-- E//"T"

select *
from ELEMENTS e, TEXTS t
where e.term = 'E'
and    t.term = 'T'
and    e.docno = t.docno
and    e.begin < t.wordno
and    t.wordno < e.end
```

(a)

```
-- E / "T"

select *
from  ELEMENTS e, TEXTS t
where e.term = 'E'  and t.term = 'T'
and     e.docno = t.docno
and     e.begin < t.wordno
and     t.wordno < e.end
and     e.level = t.level -1
```

(b)

```
-- E="T"

select *
from ELEMENTS e, TEXTS t
where e.term = 'E'
and    t.term = 'T'
and    e.docno = t.docno
and    t.wordno = e.begin + 1
and    e.end = t.wordno + 1
```

(c)

```
--  distance ("T1", "T2") <= n

select *
from TEXTS t1, TEXTS t2,
where t1.term = 'T1'
and    t2.term = 'T2'
and    t1.docno = t2.docno
and    t2.wordno > t1.wordno
and    t2.wordno <=  t1.wordno + n
```

(d)

Figure 2: Translations of basic containments to SQL.

ment the searches of other relational data and vice versa. These kinds of queries are called "Web-Supported Database Queries" and "Database-Supported Web Queries" in [15].

Suppose we have in our database an inverted index of the DBLP XML files, and a `GRADUATES` table holding data about graduate students. We can find all students who have a DBLP entry using a SQL query (see [36]).

The query would not be so straightforward if the DBLP files are indexed in a separate system while the `GRADUATES` table is kept in a database. In that case, a natural way would be to build an additional module on top of both systems to pull `GRADUATES` rows and the DBLP inverted index entries out and join them in that module. In fact, [15] is mainly concerned with the construction of this module. A join algorithm has to be implemented in it, and this duplicates features already present in the RDBMS. Furthermore, the processing in the module would be costly if there are large number of `GRADUATES` rows and inverted index entries satisfying the query. It is much more efficient to have a more tightly integrated system, because (a) it allows an optimizer to potentially push selections down in order to avoid generating a large number of intermediate results, and (b) there is no need to cross the boundary of multiple systems, therefore saving the messaging and copying overheads.

3. PERFORMANCE AND ANALYSIS

We now turn to our performance study comparing the implementations of containment queries in a special-purpose inverted list engine and in a relational database system. The goal of our study is to explore the performance differences, seek reasons for the differences, and provide insights into the strengths and weaknesses of the RDBMS on containment queries.

3.1 Experimental Settings and Methodology

3.1.1 The Hardware and the Software Platforms

	Shakespeare	DBLP	Synthetic
text size	8 MB	53 MB	200 MB
inverted index size	11 MB	78 MB	285 MB
relational table size	15 MB	121 MB	566 MB
# distinct elements	22	598	715
# distinct text words	22,825	250,657	100,000
# total elements	179,726	1,595,010	4,999,500
# total text words	474,057	3,655,148	19,952,000

Table 1: The three datasets.

We could not find a commercial system that supported containment queries on XML data, so we built our own inverted list engine. It was written in C++ and used the BerkeleyDB library [32] to store the inverted lists. Berkeley DB is a toolkit that provides access methods such as B+tree, Extended Linear Hashing, Fixed and Variable-length records, and Queues. We used its B+-tree and each inverted list was stored as a record.

We experimented with two commercial database systems, DB2 UDB v7.1 and SQLServer v7.0. In the interest of space, we only report the results on DB2. Experimental results on SQLServer were similar and confirm that our observations are not specific to a particular system. We experimented with numerous combination of settings on the RDBMSs. The results reported here were obtained using default settings, except that the buffer pool size in DB2 was set to 128 MB, and hash join was enabled. Experimenting with other settings did not alter any of the conclusions.

We experimented with RDBMS indexes using different combinations of columns as the indexing key. In the interest of space, we only report two representatives. One is the clustered index on `(term, docno)` columns, the other is a clustered index on all columns in a table. We call the former the *two-column(2col) index* and the latter the *cover index* (as the indexing key covers all columns). Note that the term "index" is overloaded in this paper. There is the IR inverted index, and there are indexes in the RDBMS. To make it clear, the term "index" without any qualification refers to the RDBMS index.

Our inverted list engine and DB2 were run on a 800 MHZ PIII machine running Linux Redhat v6.2, and SQLServer was run on a 500 MHZ PIII machine running Microsoft NTServer v4.0. The main memory sizes on the machines are 256 MB.

3.1.2 The Datasets and the Queries

Three XML datasets were used in our study. The first was a set of Shakespeare plays [8], the second was a set of DBLP bibliography files[2], and the third was a set of synthetic XML documents. The synthetic data generator first produced a random element tree. This tree was used as a template and the number of occurrences of elements and their text contents were varied to generate a document. In the synthetic data set, the numbers of occurrences of text words followed the Zipfian distribution [37] with constant 1.0, and the numbers of occurrences of elements followed the Zipfian distribution with constant 1.5.

[2]The original archive consists of 141,023 small files averaging 374 bytes each. We combined these files into bigger ones to obtain a dataset averaging 93 KB per document.

	number of term1	number of term2	result rows
QS1	90	277	2
QS2	107,833	277	36
QS3	107,833	3,231	1,543
QS4	107,833	1	1
QD1	654	55	13
QD2	4,188	712	672
QD3	287,513	6,363	6,315
QD4	287,513	3	3
QG1	50	1,000	809
QG2	134,900	55,142	1,470
QG3	701,000	165,424	21,936
QG4	50	82,712	12
QG5	701,000	17	4

Table 2: Number of occurrences of terms and number of result rows.

Queries	Inv. List	DB2 (2col ndx)	DB2 (cover ndx)
QS1	0.1	4	2
QS2	52.9	5963	259
QS3	68.4	5219	6089
QS4	53.7	23	2
QD1	0.2	7	2
QD2	3.3	545	518
QD3	202.1	76130	3907
QD4	177.6	28	2
QG1	1.5	24	21
QG2	125.1	182997	56956
QG3	542.3	1516743	21463
QG4	50.2	931	782
QG5	369.2	4292	56

Table 3: Raw timings [msec].

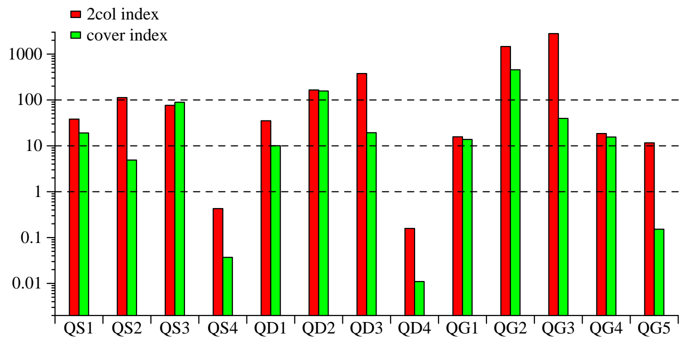

Figure 3: DB2/Inverted List performance ratios [log scale].

Statistics for the three datasets are listed in Table 1. This table shows the raw sizes of the datasets, the sizes of the inverted listes stored in BerkeleyDB B+-tree, and the DB2 relational table sizes (not including DB2 indexes). It also shows the distinct and total number of elements and text words. Note that the total number of elements and text words are the cardinalities of the `ELEMENTS` table and `TEXTS` table, respectively.

We focused on using simple queries for our performance study, as these "micro benchmark" queries allow us to study the core performance issues better than complex ones. This methodology of using simple queries rather than full workloads was indicated to be advantageous in [2]. We also experimented with complex queries, the discussion of which can be found in the full version of the paper [36]. We report results on thirteen simple containment queries of the form "`term1 contains term2`", where "`term1`" is an XML element and "`term2`" is a text word. Note that the discussion throughout this paper applies to the case where both terms are elements. In fact, the containment between elements can be processed almost identically to the containment between element and text word. This is because, since XML elements are strictly nested, whether an element is contained in another element can be checked using only the *begin* position of the nested term (`term2`).

Each query 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. Each of the two terms in a query can have a small, medium, or large number of occurrences, and we selected queries to cover most combinations. This allows us to determine how sensitive the performance is to the selectivity of terms. Table 2 shows the number of term occurrences and the number of results for each query.

The SQL version of the queries is shown in Figure 2(a). Note that each of these queries requires merging of two lists in the inverted list engine, and a join between the `ELEMENTS` table and the `TEXTS` table in the RDBMSs. Take QG3 for example. It requires merging two inverted lists of about 10 MB and 2 MB, and a join between 701,000 rows and 165,424 rows in the RDBMSs.

3.2 Results Comparing the Inverted List Engine and DB2

Table 3 shows the raw timings of the thirteen queries on the inverted list engine and DB2, all from hot runs. Figure 3 shows the performance ratios. It shows that:

1. DB2 outperforms the inverted list engine for some of the queries;

2. DB2 performs worse than the inverted list engine for most of the queries;

3. The performance using the cover index is better than using the 2col index.

The last point is easy to see as the cover index is sufficient for answering queries, whereas when the 2col index is used, rows must be fetched from the tables after the index is scanned. Unless otherwise indicated, hereafter we use the cover index performance to compare with the inverted list engine. We next investigate: (1) Why does the RDBMS sometimes perform better, and (2) Why does the RDBMS usually perform worse?

3.3 Why Does the RDBMS Sometimes Perform Better?

Recall that in the inverted list engine, each list is stored as a record, and there is no further index on it, thus there is

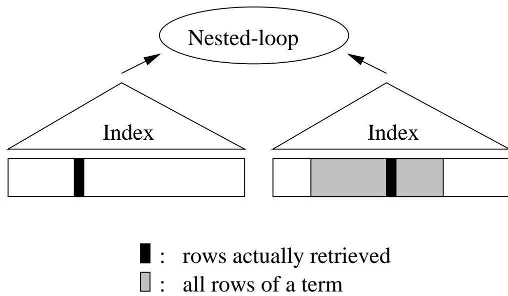

Figure 4: Index nested-loop join is used to avoid retrieving all rows of the inner term.

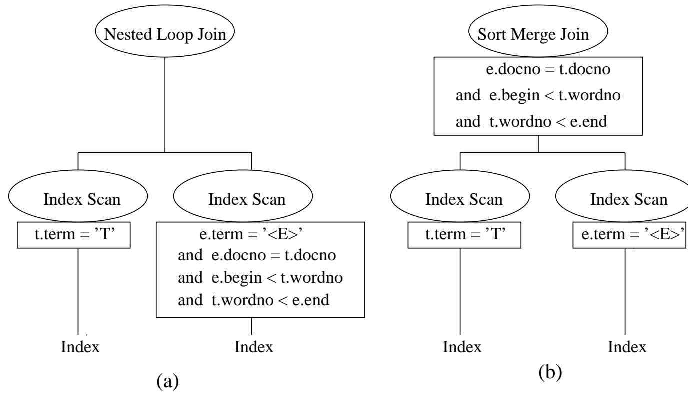

Figure 5: (a) A query plan using index nested-loop join. (b) A query plan using merge join.

no easy way to extract a portion of it other than retriving the entire list. Further, even if only a small portion of a list turns out to join with another list, all entries in the list must be examined. In the RDBMS implementation, with indexes built on the tables, rows of a term can be selectively retrieved, thus saving both I/O and CPU time.

The queries for which DB2 performs better have a common characteristic: one term in the query is highly selective, while the other term is not selective. The RDBMS optimizer is able to discern such a case, and choose the index nested-loop join between the `ELEMENTS` table and the `TEXTS` table, putting the table containing the more selective term as the outer, and the table containing the less selective term as the inner. The result is that only the inner rows satisfying the pushed-down join predicates are retrieved and joined with the outer rows. Figure 4 illustrates this situation, and Figure 5(a) shows the query plan.

There are two important points to note. First, the savings on CPU and I/O must be large enough to compensate for the overheads incurred by the RDBMS. DB2 also chose the index nested-loop join for some other queries, but it was not able to out-perform the inverted list engine for those queries. Second, we expect that, in practice, this case in which the RDBMS out-performs the inverted list engine will not be rare. It will arise whenever one predicate (or a combination of multiple predicates) in a query is highly selective; in those cases a further join on the two tables will be very efficient and an RDBMS is likely to perform better. A concrete

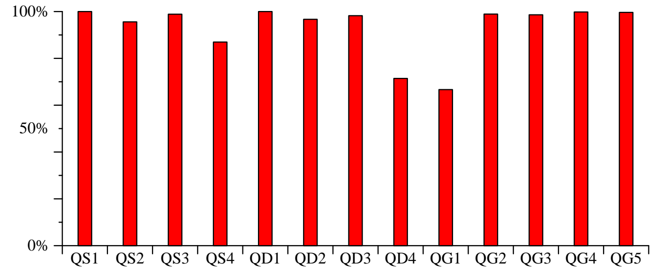

Figure 6: Percentages of CPU cost.

example is shown in [36].

3.4 Why Does the RDBMS Usually Perform Worse?

We considered a long list of possible answers to this question, including:

- The cost of binding results out of the database engine to the application might be very high, while the actual processing may be efficient;
- The optimizer may not be producing good plans;
- I/O in the RDBMS might be more expensive;
- The CPU cost might be high, due to code path, interpretive execution, overheads of locking, buffer pool management, etc..

We briefly examine each possibility in turn, and then get to our main findings.

- While the bind-out cost probably plays some role, it can be rejected as being the dominant cost because bindout is apparently a small fraction of the total execution time. To see this, note that on Query QG1 DB2 returns 809 result rows in 21 msec, whereas Query QS2 takes 259 msec to return only 36 rows. Assuming that *all* the time for QG1 is due to result bindout (clearly an overestimate), DB2 can bind out a row in $21/809 = .026$ msec. Then in QS2, returning 36 rows should take DB2 about $36 * .026 = 0.9$ msec, which is less than one percent of QS2's total execution time (259 msec.)
- Figure 5 shows two DB2 plans using the cover index and two standard joins. In general, we found that the optimizer's pick matched our own intuition of what constitute a good plan.
- Figure 6 shows the percentage of CPU cost for the thirteen queries as reported by the DB2 performance monitor. Clearly the queries are CPU-bound, and DB2 can almost fully operate out of the buffer pool with little I/O overhead.

So, if none of bind-out cost, query plan, and I/O is the dominant reason, what is causing the performance differential? To begin to answer this question we performed a deeper analysis of the algorithms used by the inverted list engine and the DBMSs.

```
procedure containment merge (list1, list2)
begin
1.  set cursor1 at beginning of list1
2.  set cursor2 at beginning of list2
3.  while (cursor1 ≠ end of list1 and
4.          cursor2 ≠ end of list2) do
5.    if (cursor1.docno < cursor2.docno) then
6.      cursor1++
7.    else if (cursor2.docno < cursor1.docno) then
8.      cursor2++
9.    else
10.     mark = cursor2
11.     while (cursor2.position < cursor1.position and
12.             cursor2 ≠ end of list2) do
13.       cursor2++
14.       if (cursor2 == end of list2) then
15.         cursor1++
16.         cursor2 = mark
17.       else if (cursor1.val (directly)contains cursor2.val) then
18.         mark = cursor2
19.         do
20.           merge cursor1 and cursor2 values
21.           cursor2++
22.         while (cursor1 value (directly)contains cursor2 value
23.               and cursor2 ≠ end of list2)
24.         cursor1++
25.         cursor2 = mark
26.       endif
27.     endwhile
28.   endif
29. endwhile
end
```

Figure 7: The inverted list containment merging algorithm.

3.5 MPMGJN: The Join Method of the Inverted List Engine

3.5.1 MPMGJN and the Standard Merge Join

Figure 7 presents the basic algorithm that does containment merging in the inverted list engine. Our actual implementation uses a slightly more optimized version. This is in fact a merge join algorithm, but is different from the standard merge join implemented in the two commercial database systems. Recall that in the SQL version of the benchmark query (Figure 2(a)), there are multiple join predicates: $t1.docno = t2.docno\ AND\ t1.begin < t2.wordno\ AND\ t2.wordno < t1.end$. The standard merge join algorithm only uses the equality join predicate on *docno* to merge two sets of rows, the inequality predicates are applied on each pair of rows with matching *docno* values. One can view this join process as two *logical* steps. In the first step, the equality predicate on *docno* is used to produce pairs of rows whose *docno* values match. In the second step, the inequality predicates are then applied on these matching rows.

On the other hand, the merge join algorithm used in the inverted list engine uses all join columns (on `docno, begin, end, wordno`) to guide merging. By doing so it is able to avoid some row comparisons done by the standard merge join. We call this merge join "Multi-predicate Merge Join"(MPMGJN). Figure 8 illustrates with an example that joins two lists. A line connecting two rows indicates an attempt to join them by doing some comparisons[3] Clearly, MPMGJN does fewer comparisons than the standard merge join.

However, the standard merge join is only one of the choices of an RDBMS. Two other algorithms can be used to process a join: hash join and index nested-loop join. Since a hash join cannot be used for inequality predicates, only the predicate on *docno* can be used, and the inequality predicates must be applied on each pair of rows with matching *docno* just like the standard merge join. Therefore, hash join has the same disadvantage. Next we look at the index nested-loop join.

3.5.2 MPMGJN and the Standard Index Nested-Loop Join

The DB2 query plan using the index nested-loop join and the cover index was shown in Figure 5(a). All three join predicates are pushed down to the inner. Figure 9(a) illustrates this join operation. For each outer row, its values (other than that on the `term` column) are used to seek the index on the inner table, starting from the root of the B+-tree and reaching a record with the *start key* at the bottom of the index. An index scan is then conducted across the index records until one with a *stop key* is reached. Then each record along the scan is attempted to be joined with the outer row. The seeking and scanning are repeated for all outer rows.

Here we use "start key" and "stop key" to refer to points where the index scan can start and stop. An RDBMS uses "start key predicates" (or "sargable predicates") and "stop key predicates" to find these points. For our containment queries, the start and stop key predicates are as follows. When the outer table is `ELEMENTS`, inner table is `TEXTS`, the start key predicates are: $term = value\ AND\ docno = outer.docno\ AND\ wordno < outer.begin$; the stop key predicates are: $term = value\ AND\ docno = outer.docno\ AND\ wordno < outer.end$. When the outer table is `TEXTS`, the inner table is `ELEMENTS`: the start key predicates are: $term = value\ AND\ docno = outer.docno\ AND\ end > outer.wordno$; the stop key predicates are: $term = value\ AND\ docno = outer.docno\ AND\ begin < outer.wordno$.

Using the start and stop keys, an RDBMS is able to selectively retrieve and examine inner rows to join with outer rows. Figure 8(c) illustrates with the same example. Again, a line between two rows indicates an attempt to join them by doing comparisons[4]. It appears from this example that the standard index nested-loop join does fewer comparisons than MPMGJN, and therefore should perform better. This is not always true.

In order to selectively examine inner rows, an index must be used, and comparisons must be done on index records. We call the process of going through the index to retrieve the inner rows *index seek*, and the process of joining inner and outer rows *index scan*. We call the number of comparisons done during the index seek the *index seek length*, and the number of comparisons done during the index scan the *index scan length*.

To see the index seek overhead, let us do a simple calcula-

[3]In the inverted list engine, the predicates are implemented as "short-circuit" conditions. That is, if one predicate fails, the other predicates will not be applied.

[4]For each outer, one more comparison is done to determine when to stop.

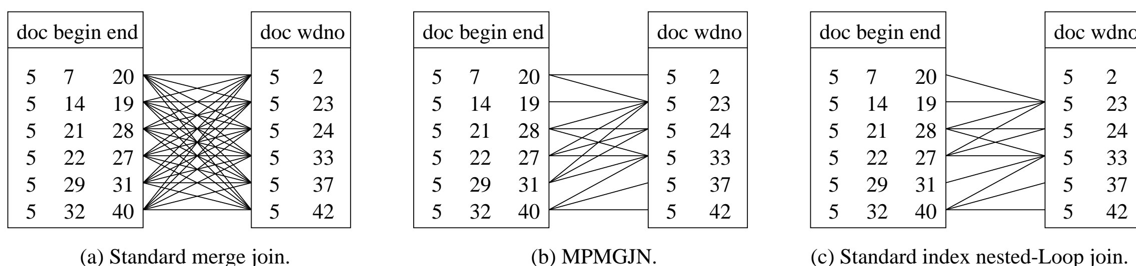

Figure 8: Workout of an example.

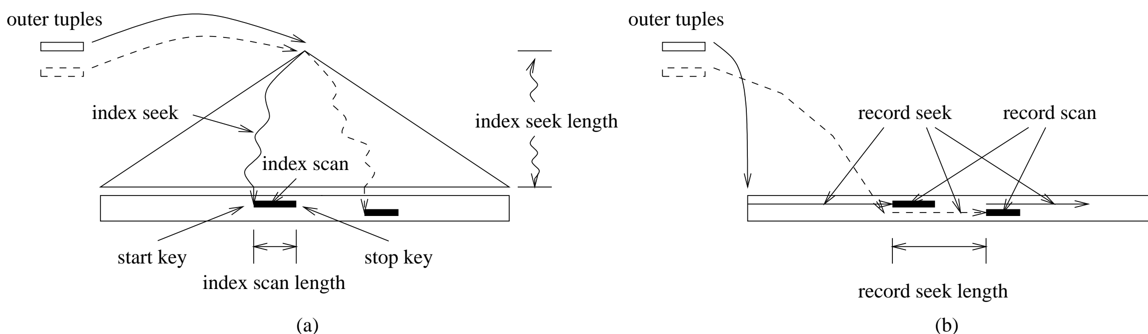

Figure 9: The operation on the inner of: (a)index nested-loop join, (b) merge join, as a form of nested-loop join.

tion. Assume that the index has 4 levels[5], each index page holds 256 keys, and binary search is used to find the pointer to follow to a next level index page. The index seek length in the worst case is $4 \times \log 256 = 32$, and in the average case is $32/2 = 16$. This means that each outer row must pay 16 more comparisons to get some inner rows to join with it.

Further, these additional comparisons are costly due to their poor hardware cache utilization. Let us review the memory hierarchy in a modern computer system. Data and instructions higher in the memory hierarchy can be accessed much faster than those in a lower hierarchy. In our machine, the first level cache can be accessed in one cycle, but a miss costs many cycles and thus is much more costly. If the second level cache is missed, the penalty is even higher. A cache consists of multiple cache lines, each of which can hold multiple instructions or data. When a datum is missing in a cache, it is fetched from the lower memory hierarchy, and data residing in the same cache line are also brought in. If there is prefetching, subsequent data may also be brought in to fill other cache lines. Thus the access to the nearby or subsequent data is likely to result in a hit and be fast. However, this benefit only exists when data is accessed sequentially or when the access pattern can be detected by the processor. Random access is likely to result in more cache misses than sequential access.

Binary search is very efficient in reducing the number of comparisons during index seek, however it guarantees almost no access of contiguous records. Therefore an index record comparison almost always incur a cache miss.

A merge join (MPMGJN is no exception) is essentially a form of nested-loop join, except that seeking is not done on an index, but rather directly on data records. This "nested-loop" join is performed in the following way: for each outer row, a seek is done on the inner rows until a "start record" is found, then a record scan is conducted and each row during the scan is attempted to join with the outer row; the record scan ends at a "stop record". The next seek does not need to start from the first record, but instead can start from the beginning of last record scan. A merge join can be done this way because both the inner rows and the outer rows are sorted. Figure 9(b) illustrates the operation on the join inner. A *record seek* is analogous to an index seek, and a *record scan* is analogous to an index scan.

For the same query on the same data, record scans (note that each outer row requires a seek and a scan) cost the same number of comparisons as index scans, but the record seek costs are different from the index seek costs. A merge join has better cache utilization as both the outer and the inner rows are, by and large, retrieved and examined sequentially. In fact, it is not 100% straight sequential because some inner rows may need to be looped over multiple times. This looping increases the possibility of cache hits. There is a disadvantage on record seeks however, a record seek may take more comparisons than an alternative index seek.

3.5.3 Experimental Comparisons of Join Algorithms

To see the performance impact of the join algorithms, we implemented the standard merge join and index nested-loop join ourselves and compare them with MPMGJN. As described in Section 3.5.1 the standard merge join implementation applies the inequality join predicates on every pair of rows whose *docno* values match.

To emulate the standard index nested-loop join, we con-

[5]This is the height of some of our DB2 indexes.

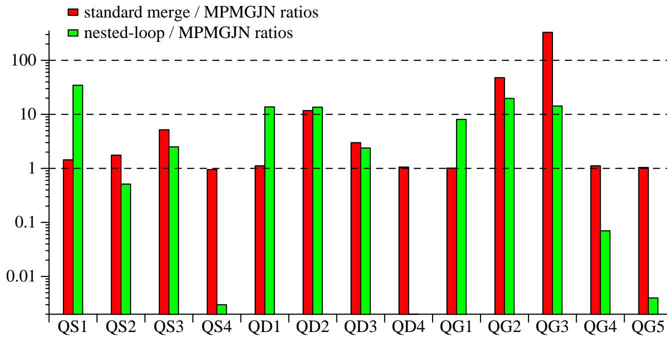

Figure 10: Ratios of standard joins to MPMGJN [log scale].

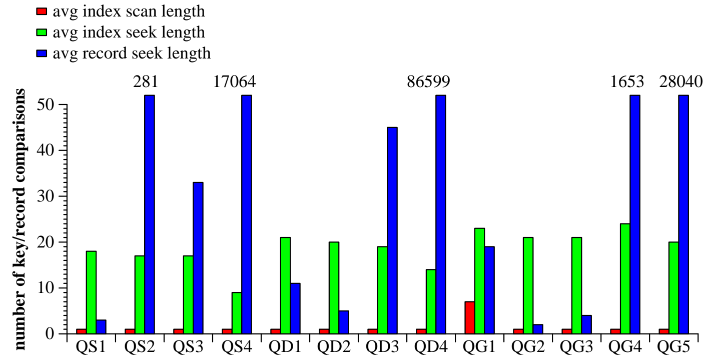

Figure 11: Seek lengths and scan lengths, averaged per outer.

Queries	MPMGJN	standard merge join
QS1	5	1,653
QS2	7,131	984,948
QS3	89,716	10,175,904
QS4	2,366	3,475
QD1	503	555
QD2	4,723	1,315,662
QD3	263,458	14,082,080
QD4	1,766	4,950
QG1	1,000	1,000
QG2	103,994	148,773,116
QG3	610,816	2,319,244,480
QG4	12	82,712
QG5	56,084	238,340

Table 4: Number of row pairs compared by MPMGJN and the standard merge join.

verted our inverted lists into relational rows according to the schema presented in Section 2.3 and stored them in a BerkeleyDB B+-tree, making it equivalent to the DB2 cover index. Our index nested-loop join implementation simulates the optimizer by choosing the table containing the shorter list as the outer. Start and stop key predicates were used.

In our experiments, the running times of each of the three algorithms include (a) the parsing cost to extract individual column values from records, and (b) the cost to generate results, although the printing cost is not included. We assume that data are cached in the buffer pool (thus no I/O). This is a reasonable assumption as we have shown that the queries are CPU-bound and the I/O cost is minimal.

Figure 10 shows the performance ratios of the two standard join algorithms to MPMGJN. Note that the Y-axis is again in log-scale. These results confirm that the MPMGJN performs at least as well as the standard merge join, and better than the index nested-loop join for most queries. The index nested-loop join performs better than the MPMGJN for queries QS2, QS4, QD4, QG4, and QG5. DB2 used the index nested-loop join for these queries except QG4, and did perform better than the inverted list engine for QS4, QD4 and QG5 (see Figure 3).

To better understand why MPMGJN can perform better than the standard merge join, Table 4 shows the number of row pairs compared by the two merge algorithms.

Figure 11 compares index/record seek lengths, and index/record scan of the queries. Since different outer rows may require different seek and scan lengths, for each query we obtained the averages over all outer rows. Figure 11 shows that long seeks are performed for short scans for all queries. For six of them, QS1, QD1, QD2, QG1, QG2, and QG3, record seek lengths are shorter than index seek lengths. These are precisely the queries that the MPMGJN out-performs the index nested-loop join by a wide margin (Figure 10). For the rest seven queries, record seek lengths are longer than index seek lengths, and the difference for five of them are dramatic. These five queries, QS2, QS4, QD4, QG4, and QG5, are precisely those for which the index nested-loop join out-performs the MPMGJN. For the rest two queries with record seek lengths longer than index seek lengths, MPMGJN still performs better because of sequential scan. This demonstrates that sequential scan is better than random access, but only until the amount of extra work done exceeds a certain point.

3.6 Hardware Cache Utilization

With main memory sizes getting larger and the memory hierarchy getting deeper, many researchers (e.g., [2, 31, 25]) have recognized the effect of good hardware cache utilization on performance. We conducted experiments to see whether there exists significant difference between the inverted list engine and the RDBMS on containment queries. We show the results in this section.

This set of experiments were conducted on a machine with a 800 MHZ Intel PIII processor running a v.2.2.16 Linux kernel. The machine has a 16 KB first level instruction cache (L1-I cache), a 16 KB first level data cache (L1-D cache), and a 256 KB second level (L2) unified cache. The data and instruction accesses to the second level cache can be measured separately. The PIII processor supports prefetching and out-of-order instruction execution. The latter implies that instruction/data fetches can be overlapped. The detailed description of the hardware parameters can be found in [18]. We measured the number of cache accesses and misses using the PIII hardware counters.

In the interest of space, we discuss the hardware cache utilization of three of the queries (QD3, QD4 and QG1). Figures 12-14 show the number of accesses and misses in thousands. Overall, the number of L1 cache accesses is significantly (up to $30,000$ times) larger than the L2 cache ac-

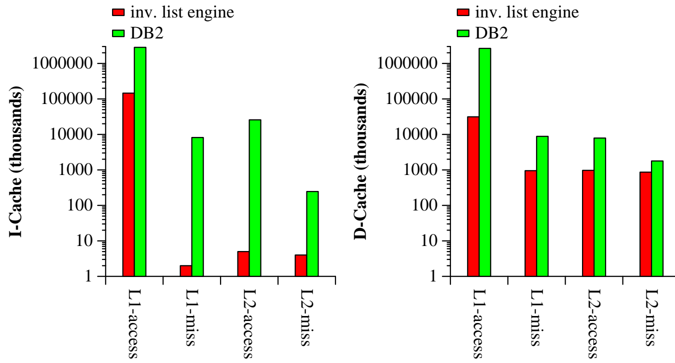

Figure 12: Query QD3 [log scale]

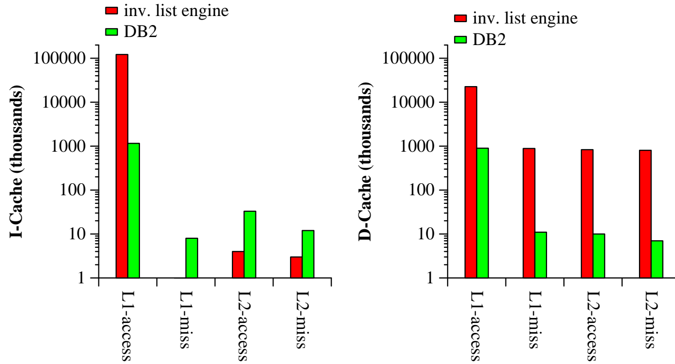

Figure 13: Query QD4 [log scale]

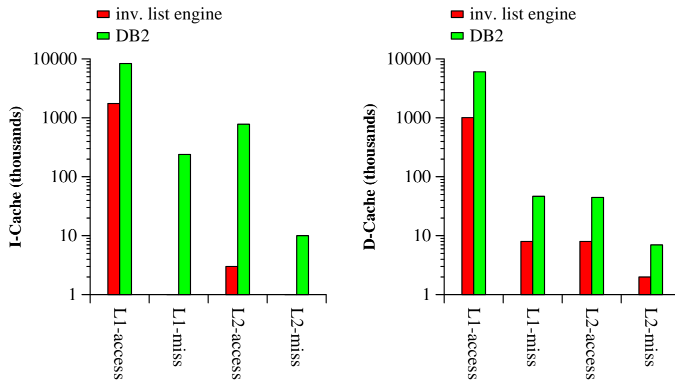

Figure 14: Query QG1 [log scale]

cesses. This is because the L2 cache is accessed only when the L1 caches are missed. Notice that for data caches, the number of L1 misses is the same as the number of L2 accesses, while for instruction caches, the number of L1 misses is less than the number of L2 accesses. This reveals instruction prefetching but not data prefetching. Next we examine each query in turn.

QD3 is a query on which DB2 chooses index nested-loop join, and performs much worse than the inverted list engine (Figure 3). The inverted list engine has about 19 times fewer accesses to the L1-I cache and about 84 times fewer accesses to the L1-D cache compared to DB2. Further, the inverted list engine has over $4,000$ times fewer misses in the L1-I cache, and about 9 times fewer misses in the L1-D cache (thus DB2 actually achieves better data cache miss ratio).

QD4 is a query on which DB2 chooses the index nested-loop join as well, and this is one of the queries on which DB2 out-performs the inverted list engine. It is clear from Figure 13 that DB2 wins in most aspects: it accesses the caches less often and also misses less (with some exceptions in the instruction cache). From Figure 11 we see that the record seek length is dramatically larger than the index seek length for this query. This means that the inverted list engine has to seek through a large number of inner rows to find some that join with the outer rows. DB2 on the other hand, uses the index to find those rows and does much less work.

QG1 is a query on which DB2 chooses the standard merge join. Both join algorithms happen to do the same number of comparisons for this query (Table 4), and our own implementation also indicates that the two algorithms should perform the same (Figure 10). However, the inverted list engine performs more than an order of magnitude faster than DB2. Figure 14 gives us insight into the cause of the difference. As we can see, DB2 has about five times as many accesses to both the L1-I cache and the L1-D cache. Further, it has about 240 times as many misses in the L1-I cache and about 6 times as many misses in the L1-D cache. For the L2 cache, DB2 has over 5 times more misses.

For query QG1, DB2 has better data cache miss ratio, but much worse instruction cache miss ratio. In addition, even though the algorithm indicates the same amount of work, the cache utilization of the two systems are different, therefore cache is a distinct factor that affects performance.

4. RELATED WORK

A substantial amount of 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 [4, 35, 10]. Commercial examples include the DB2 Text Extender [16], SQL Server Full-Text Search Service [34] and Oracle InterMedia Text [23]. An example of integrating text search with semi-structured databases is Lore [20], in which a simplified version of an IR-style text index is used to locate strings containing specific text words or groups of text words [19]. None of this previous work explores the performance implications of a special purpose vs. native implementation of this functionality in an RDBMS.

The advent of SGML [13] has triggered much research on integrating content and structure in text retrieval, including [3, 33, 4, 26]. Work on containment queries can be found in [6, 7, 9]. Our work on containment queries differs from the previous work in that, since we target XML rather

than SGML data retrieval, and XML elements are strictly nested, we are not concerned with overlapped extents, nor with reduction functions on overlapped extents. Most significantly, our work does not focus on the development of containment algorithms; rather, it focuses on how to implement the algorithms in an RDBMS.

There is also work that considers using an RDBMS to store and retrieve XML documents, including [29, 30, 11]. These papers focus on techniques for converting XML documents to and from relations and are complementary to our work, which focuses on the performance of implementation options for a class of query over XML data.

Putz [24] describes using a relational database system for information retrieval. His work differs from ours in the relational schema for, and hence the retrieval of, the inverted index. In [24], multiple encoded occurrences of a term are stored in one relational row, therefore the storage is more compact. The flip side is that the application program has to do quite a bit of work, such as encoding/decoding and packing/unpacking table rows, and doing operation on postings retrieved. Thus the power and flexibility of the RDBMS is not fully utilized, as the IR and "normal" query processing is not really integrated. Also, [24] does not consider structured text retrieval, and does not compare the RDBMS with the IR approach.

Florescu et al. [12] use a different schema to store the inverted index, where the postings of each distinct XML element and text word is stored in its own table. Thus we would have a `LINE` table for "<line>", a `CLEOPATRA` table for "cleopatra". The implementation options we discuss apply equally well to this model; needless to say, using this model many tables would be required to store the inverted index. Also, [12] does not compare the implementation on the RDBMS with the IR alternative.

5. CONCLUSION

While the dust has not yet settled on the debate over which XML query language will win, or what role RDBMSs will eventually play in XML query systems, two things are clear. First, containment queries will be an important part of XML query workloads. Second, at least in the foreseeable future, a great deal of XML data will be stored in relational systems. Currently in the commercial world the approach in which an IR indexing system is "glued" to a relational system, dominates.

However, as we have argued in the introduction, there are compelling reasons to consider a more tightly coupled approach, in which queries involving containment are supported by native RDBMS data structures, query optimizers, and query processors. This tightly coupled approach will not be viable unless its performance is satisfactory. Our experiments show that with current commercial RDBMS technology, in general a native RDBMS implementation of containment query support is substantially slower than that of a special purpose IR engine. We sought to quantify this performance differential, and to gain insights as to whether the situation could be remedied.

There appears to be no single factor that accounts for the entire performance difference between the two types of systems, and modifying an RDBMS so that its performance matches that of the special-purpose inverted list engine will be non-trivial. However, we have discovered two important contributing factors to the performance difference, these are the join algorithm used by the inverted list system, which we call multi-predicate merge join (MPMGJN), and the hardware cache utilization achieved.

As we have demonstrated, for joins generated by containment queries, the MPMGJN algorithm can be more than an order of magnitude faster than standard RDBMS join algorithms. It appears that the addition of this new join algorithm will be a critical part of any successful effort to make an RDBMS competitive with a special purpose IR engine on XML containment queries. In addition, in our experiments the RDBMS had much lower cache utilizations than the IR systems. The on-going research on main memory and cache aware database systems is likely to produce a new generation of RDBMSs that have much better cache utilization. While it is premature to make concrete predictions, we are optimistic that by combining better join algorithms with better cache utilization, an RDBMS will be able to natively support containment queries efficiently.

6. ACKNOWLEDGMENTS

The authors would like to thank Anastassia Ailamaki for her help in measuring hardware cache utilization. Funding for this work was provided by NSF through grants CDA-9623632 and ITR 0086002, and DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908.

7. REFERENCES

[1] S. Abiteboul, D. Quass, J. McHuge, J. Widom, and J. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[2] A. Ailamaki, D.J.DeWitt, M.D.Hill, and D.A.Wood. Dbmss on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, September 1999.

[3] Ricardo Baeza-Yates and Gonzalo Navarro. Integrating contents and structure in text retrieval. *SIGMOD Record*, 25(1):67–69, March 1996.

[4] G. E. Blake, M. P. Consens, P. Kilpelainen, P.-A. Larson, T. Snider, and F. W. Tompa. Text/relational database management systems: Harmonizing sql and sgml. In *Proceedings of the International Conference on Applications of Databases*, pages 267–280, June 1994.

[5] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. *XQuery: A Query Language for XML*. W3C Working Draft, February 2001. http://www.w3.org/TR/xquery.

[6] C. L. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

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

[8] Robin Cover. The xml cover pages. In *http://www.oasis-open.org/cover/xml.html*, July 2000.

[9] Tuong Dao, Ron Sacks-Davis, and James A. Thom. Indexing structured text for queries on containment

relationships. In *Proceedings of the 7th Australasian Database Conference*, 1996.

[10] Stefan Dessloch and Nelson Mattos. Integrating sql databases with content-specific search engines. In *Proceedings of the 23rd VLDB Conference*, 1997.

[11] Daniela Florescu and Donald Kossman. Storing and querying xml data using an rdbms. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[12] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into xml query processing. *WWW9/Computer Networks*, 33(1-6):119–135, 2000.

[13] International Organization for Standardization. Information processing–text and office systems–standard generalised markup language (sgml), iso/iec 8879, 1986.

[14] GMD. Gmd-ipsi xql engine. http://xml.darmstadt.gmd.de/xql/index.html, August 1999.

[15] Roy Goldman and Jennifer Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *Proceedings of the 2000 Sigmod Conference*, pages 285–296, 2000.

[16] IBM. Db2 text extender. http://www-4.ibm.com/software/data/db2/extenders/text.htm, July 2000.

[17] INRIA. Minixyleme project. http://www-rocq.inria.fr/∼aguilera/xoql/minixyleme/readme.html.

[18] Intel. Intel architecture software developer's manual, volume 1: Basic architecture, 1999.

[19] J.McHugh, J.Widom, S.Abiteboul, Q.Luo, and A.Rajaraman. Indexing semistructured data. In *Stanford Technical Report*, January 1998.

[20] J.McHugh, S.Abiteboul, R.Goldman, D.Quass, and J.Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[21] University of Washington. The tukwila data integration system. http://data.cs.washington.edu/integration/tukwila/.

[22] University of Wisconsin. The niagara system. http://www.cs.wisc.edu/niagara/.

[23] Oracle. Oracle8i intermedia text reference, release 8.1.5. http://oradoc.photo.net/ora81/DOC/inter.815/a67843/toc.htm.

[24] Steve Putz. Using a relational database for an inverted text index. In *Xerox Palo Alto Research Center Technical Report SSL-91-20, Xerox PARC*, January 1991.

[25] Jun Rao and Kenneth A. Ross. Making b+-trees cache conscious in main memory. In *SIGMOD Conference*, pages 475–486, 2000.

[26] Ron Sacks-Davis, Timothy Arnold-Moore, and Justin Zobel. Database systems for structured documents. In *Proceedings of the International Symposium on Advanced Database Technologies and Their Integration (ADTI'94)*, pages 272–283, October 1994.

[27] Arnaud Sahuguet. Kweelt. http://db.cis.upenn.edu/Kweelt/.

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

[29] Jayavel Shanmugasundaram, He Gang, Kristin Tufte, Chun Zhang, David DeWitt, and Jeffrey Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the 1999 VLDB Conference*, September 1999.

[30] Jayavel Shanmugasundaram, E. Shekita, R.Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as xml documents. In *VLDB Conference*, September 2000.

[31] A. Shatdal, C. Kant, and J.F.Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th VLDB Conference*, 1994.

[32] Sleepycat Software. The berkeley database. http://www.sleepycat.com.

[33] T.Arnold-Moore, M.Fuller, B.Lowe, J.Thom, and R.Wilkinson. The elf data model and sgql query language for structured document databases. In *Proceedings of the Australasian Database Conference, Adelaide, Australia*, pages 17–26, 1995.

[34] Rozanne Whalen. Implementing the full-text search service in sql server. http://msdn.microsoft.com/library/periodic/period00/ewn0092.htm, September 2000.

[35] Tak W. Yan and Jurgen Annevelink. Integrating a structured-text retrieval system with an object-oriented database system. In *VLDB Conference*, September 1994.

[36] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. full version. http://www.cs.wisc.edu/niagara/papers/ZND+01full.pdf, 2001.

[37] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Reading MA, 1949.