A Tale of Two Approaches: Query Performance Study of XML Storage Strategies in Relational Databases

Sandeep Prakash Sourav S. Bhowmick

School of Computer Engineering,
Nanyang Technological University, Singapore
assourav@ntu.edu.sg

Abstract. Several recent papers have investigated a relational approach to store XML data and there is a growing evidence that *schema-conscious* approaches are a better option than *schema-oblivious* techniques as far as query performance is concerned. This paper studies three strategies for storing XML document including one representing schema-conscious approach (SHARED-INLINING) and two representing schema-oblivious approach (XParent and SUCXENT++). We implement and evaluate each approach using benchmark *non-recursive* XQueries. Our analysis shows an interesting fact that schema-conscious approaches are not always a better option than schema-oblivious approaches! In fact, it is possible for a schema-oblivious approach (SUCXENT++) to be faster than a schema-conscious approach (SHARED-INLINING) for 55% of the benchmark queries (the highest observed factor being 87.8 times). SUCXENT++ also outperforms XParent by up to 1700 times.

1 Introduction

Recently, there has been a substantial research effort in storing and processing XML data using relational backends. This approach either shred the XML documents into relational tables using some sort of encoding scheme [1, 5, 7, 8, 12, 15, 18, 19] or use a BLOB column to store the XML document [6, 20, 21]. BLOB-based approaches use stored procedures to invoke an external XPath/XQuery processor. The main advantages of this approach are fast retrieval of full XML document and ability of storing any XML document irrespective of the availability of the schema. However, in this paper, we focus on shredding-based approaches as in the BLOB-based approaches, usually the entire XML document must be brought into memory before processing, severely limiting the size of the data and optimization possibilities.

The shredding-based approaches can be classified into two major categories: the *schema-oblivious* and the *schema-conscious* approaches. In brief, the *schema-oblivious* method consists of a fixed schema which is used to store XML documents. This approach does not require existence of an XML schema/DTD. Some examples of schema-oblivious approaches are Edge approach [7], XRel [18], XParent [8], SUCXENT++[12]. The *schema-conscious* method, on the other hand, derives a relational schema based on the DTD/XML schema of the XML documents. Examples of such approaches are SHARED-INLINING [15] and LegoDB [1]. Once XML data is stored using either schema-conscious or schema-oblivious approach, an XQuery/XPath is

translated into SQL for evaluation. A comprehensive review of methods for XML-to-SQL query translation and their limitations is beyond the scope of this paper and can be found in [9].

In this paper, we study the performance of schema-conscious and schema-oblivious approaches and compare these alternatives. In particular, we compare the performance of one schema-conscious approach (SHARED-INLINING [15]) with two representative schema-oblivious approaches (XParent[8] and SUCXENT++[12]). At this point, one would question the justification of this work as a growing body of research suggests that schema-conscious approaches perform better than schema-oblivious approaches. After all, it has been demonstrated in [16] that schema-conscious approaches generally perform substantially better in terms of query processing and storage size. However, we believe that the superiority of schema-conscious approaches (such as SHARED-INLINING) as demonstrated in [16] may not hold anymore! This is primarily due to the following reasons.

The Edge approach[7] was used in [16] as the representative schema-oblivious approach for comparison. Although the Edge approach is a pioneering relational approach, we argue that it is not a good representation of the schema-oblivious approach as far as query processing is concerned. In fact, XParent [8] and XRel [18] have been shown to outperform the Edge approach by up to 20 times, with XParent outperforming XRel [8, 11]. This does not necessarily mean that XParent outperforms schema-conscious approaches. However, it does raise the question whether there exists a schema-oblivious approach in the literature that can outperform XParent significantly and consequently outperform a schema-conscious approach? In [12], we provided answer to this question by presenting a novel schema-oblivious approach called SUCXENT++ (**S**chema **U**n**c**onscious **X**ML **En**abled Sys**t**em) that outperforms XParent by up to 15 times and a schema-conscious approach (Shared-Inlining) by up to 8 times for certain types of *recursive XML queries*. A recursive XML query is an XML query which contains the descendant axis (//) [10].

Although our effort in [12] asserts that it is indeed possible for a schema-oblivious approach to outperform the schema-conscious approach, it was demonstrated only for recursive XML queries. Naturally, we would like to know whether this result holds for *non-recursive* XML queries. Note that a non-recursive XML query does not contain any descendant axis. Such queries are prevalent in GUI-based XML query formulation framework in the presence of DTDs/XML schemas. *In other words, is it possible for* SUCXENT++ *to outperform XParent and* SHARED-INLINING *for non-recursive XML queries as well?* In this paper, we provide the answer to this question.

2 Related Work

One of the earliest work on performance evaluation of XML storage strategies was carried out by Tian *et al.* [16]. In this paper five strategies for storing XML documents were studied including one that stores documents in the file system, three that uses a relational database system (Edge and Attribute approaches [7], SHARED-INLINING [15]), and one that uses an object manager. Recently, Lu *et al.* [11] reported results on benchmarking six relational approaches on two commercial RDBMS and two native XML database systems using the XMark[14] and XMach[2] benchmarks.

Both these efforts showed that schema-conscious approaches perform better than schema-oblivious approaches. Our study differs from the above efforts in the following ways. First, we consider a relatively more efficient schema-oblivious storage strategy (SUCXENT++) compared to the approaches used in [16, 11] for our study. Second, we evaluate the performance of our representative systems on much larger data sets (1GB) having richer variety (data and text-centric single and multiple documents). Note that the maximum size of the data set was 140 MB and 150MB in [16] and [11], respectively. This gives us a better insight on the scalability of the representative approaches. Third, we experimented with wider and richer variety of XML queries. Finally, contrary to the insight gained by Tian *et al.* and Lu *et al.*, we show that it is indeed possible for a schema-oblivious approach to outperform a schema-conscious approach for certain types of non-recursive XML queries.

XMach-1 [2] is a multi-user benchmark that is based on a Web application and considers text documents and catalog data. The goal of the benchmark is to test how many queries per second a database can process at what cost. Additional measures include response times, bulk load times and database or index sizes. XMark [14] is a single-user benchmark. The database model is based on the Internet auction site and therefore, its database contains one large XML document with text and non-text data. XOO7 [3] was derived from OO7 [4], which was designed to test the efficiency of object-oriented DBMS. Besides mapping the original queries of OO7 into XML, XOO7 adds some specific XML queries. Workloads of the above benchmarks, cover different functionalities, but leave out a number of XQuery features [17]. XBench [17] was recently proposed to cover all of XQuery functionalities as captured by XML Query Use Cases. As our work is based on XBench data set and queries, we also cover all these functionalities. In summary, all of these are application benchmarks and measure the overall performance of a DBMS. That is, they compare the performance of different RDBMSs (e.g., IBM DB2, SQL Server in XBench) and native XML DBMS (e.g., X-Hive) for processing XML data. On the contrary, we focus on evaluating performance of schema-oblivious and schema-conscious approaches on a *specific* commercial RDBMS.

3 Background

In this section, we present the framework for our performance study. We begin by identifying the representative schema-oblivious and schema-conscious systems chosen for our performance study and justify their inclusion. Then, we present the experimental setup and data sets used for our study.

3.1 Representative Systems

We chose XParent[8], SUCXENT++[13], and SHARED-INLINING [15] as representative shredding-based approaches for performance study. The reasons are as follows.

First, we intend to ensure that the implementation of our selected storage scheme does not require modification of the relational engine. This is primarily because such approach enhances portability and ease of implementation of the storage approach on top of an off-the-shelf commercial RDBMS. Consequently, we did not choose the dynamic intervals approach [5] and the approach in [19] as these approaches enhance

	SD (Single Document)	MD (Multiple Document)
TC (Text-centric)	Online dictionaries	Digital libraries, news corpus
DC (Data-centric)	E-commerce catalogs	Transactional data

(a) Data set

Data set	No of Nodes		
	10MB	100MB	1GB
DC/MD	219,382	2,183,331	23,821,115
DC/SD	238,260	2,394,886	24,810,315
TC/MD	229,258	2,335,180	23,704,294
TC/SD	279,004	2,765,209	28,419,013

(b) Data set of XBench

Fig. 1. Data set of XBench.

the relational engine with XML-specific primitives for efficient execution. Note that in the absence of such XML-specific primitives, the query processing cost can be expensive in these approaches. For instance, without the special relational operators defined in [5], the query performance is likely to be inferior even for simple path expressions [9].

Second, the selected approach must have good query performance. Jiang *et al.* [8] showed that XParent outperforms Edge [7] (up to 20 times) and XRel [18] approaches significantly. In [12], we have shown that SUCXENT++ outperforms XParent (up to 15 times) and SHARED-INLINING [15] (up to 8 times) for certain types of recursive XML queries. Moreover, XParent takes 2.5 times more storage space compared to the SUCXENT++ approach. Hence, XParent and SUCXENT++ are chosen as representatives of the schema-oblivious approach.

Finally, we chose SHARED-INLINING over LegoDB [1] as the representative of schema-conscious approaches for the following reasons. First, SHARED-INLINING is widely used in the literature as a representative of schema-conscious approaches. Second, unlike SHARED-INLINING , LegoDB is application and query workload-dependent. Third, despite our best efforts (including contacting the authors), we could not get the source code of LegoDB.

3.2 Experimental Setup

Prototypes for SUCXENT++, XParent and SHARED-INLINING were developed using Java JDK 1.5 and a commercial RDBMS[1]. The experiments were conducted on a P4 1.4GHz machine with 256MB of RAM and a 40GB (7200rpm) IDE hard disk. The operating system was Windows 2000 Professional.

The XBench [17] data set was used for comparison of storage size, insertion and extraction times, as it provides a comprehensive range of XML document types. We studied both data-centric and text-centric applications consisting of single as well as multiple XML documents. We also test for scalability (small (10MB), normal (100 MB) and large (1 GB) dataset) of the schema-conscious and schema-oblivious approaches. Figures 1(a) and 1(b) summarize the characteristics of the data sets used. In our experiments, we create separate database instances for all the scenarios. For example, we create three database instances for TC/SD, called TC/SD-small, TC/SD-normal, and TC/SD-large. A total of 22 queries as described in [17] were tested on this data set. The list of queries and their characteristics is shown in Figures 2 and 3. *Observe that we focus on non-recursive XML queries.*

The queries were executed in the *reconstruct* mode where not only the internal nodes are selected, but also all descendants of that node. Several steps were taken to

[1] Our licensing agreement disallows us from naming the product.

#	Query	Characteristics
1	for $order in input()/order[@id="1"] return $order/customer_id	- DCMD - Exact match
2	for $a in input()/order[@id="2"] return $a/order_lines/order_line[1]	- DCMD - Exact match, ordered access
3	for $a in input()/order[@id="4"] return $a//item_id	- DCMD - Exact match - One '//' axis
4	for $a in input()/order where $a/total gt 11000.0 order by $a/ship_type return <Output> {$a/@id}{$a/order_date}{$a/ship_type} </Output>	- DCMD - Sort - Return multiple elements
5	for $a in input()/order[@id="6"] return $a	- DCMD - Exact match - Document reconstruction
6	for $order in input()/order, $cust in input()/customers/customer where $order/customer_id = $cust/@id and $order/@id = "7" return <Output>{$order/@id}{$order/ order_status}{$cust/first_name}{$cust/ last_name}{$cust/phone_number}</Output>	- DCMD - Join - Multiple return elements

#	Query	Characteristics
7	for $item in input()/catalog/ item[@id="I1"] return $item	- DCSD - Exact match
8	for $item in input()/catalog/:item where every $add in $item/authors/author/ contact_information/mailing_address satisfies $add/name_of_country = "Canada" return $item	- DCSD - Quantification
9	for $a in input()/catalog/item[@id="I6"] return <Output> {$a/authors/author[1]/ contact_information/mailing_address} </Output>	- DCSD - Document construction
10	for $a in input()/catalog/:item where $a/date_of_release gt "1990-01-01" and $a/date_of_release lt "1991-01-01" and empty($a/publisher/ contact_information/FAX_number) return <Output> {$a/publisher/name} </Output>	- DCSD - Data type casting - Irregular data

Fig. 2. XBench queries.

ensure the consistency of the test results. Prior to our experiments, we ensure that statistics had been collected, allowing well-informed plan selection. Each test query was executed 6 times while the performance results of the first run discarded. A 95% confidence interval was computed. In our experiments, the estimated error was small ($< 10\%$). The bufferpool of the DBMS was cleared before each run to ensure times reported are from cold runs. Also, appropriate indexes were constructed for all the three approaches through a careful analysis on the benchmark queries. Note that the RDBMS we have used can only create an index for *varchar*, which is less than 255. Hence, in schema-oblivious approaches, if a single text in the "value" attribute exceeds the limit, which is highly possible for text-centric XML documents, then we cannot create an index on this attribute to facilitate XML query processing. Similar to [11], we use a two-table approach to handle short/long data values separately.

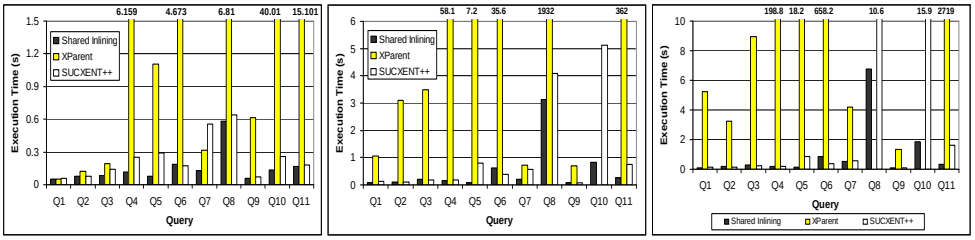
4 Data-Centric Query Processing

In this section we study the performance of the representative approaches (Shared-Inlining , Sucxent++, and XParent) for XML queries on data-centric XML documents. The reader may refer to [15], [12, 13], and [8] for algorithmic details related to XML query translation and evaluation. Note that we do not discuss storage size, document insertion and extraction performance here as we have already presented this in [13]. In general, Sucxent++ is 5.7 - 47 times faster than XParent and marginally better than Shared-Inlining as far as insertion time is concerned. Sucxent++ is also the most efficient among the three approaches as far as document extraction is concerned. On the other hand, Shared-Inlining requires the least amount of storage. XParent takes 2.5 times more storage space compared to Sucxent++.

We use the queries Q1 to Q11 in Figures 2 and 3 for our performance study. The results are shown in Figure 4. The maximum time we allowed a query to run for all

#	Query	Characteristics
11	for $size in input()/catalog/:item/ attributes/size_of_book where $size/length*$size/width*$size/ height > 500000 return <Output> {$size/../../title} </Output>	- DCSD - Data type casting
12	for $prolog in input()/article/prolog where $prolog/authors/author/name="Ben Yang" return $prolog/title	- TCMD - Exact match
13	for $a in input()/article[@id="8"]/body/ section[@heading="introduction"] return <HeadingOfSection> {$a/@heading} </HeadingOfSection>	- TCMD - Two conditions
14	for $a in input()/article where some $b in $a/body/abstract/p satisfies (contains($b, "the") and contains($b, "hockey")) return $a/prolog/title	- TCMD -Quantification
15	for $a in input()/article[@id="5"] return <Output> {$a/prolog/title} {$a/prolog/authors/author/name} {$a/prolog/dateline/date} {$a/body/abstract} </Output>	- TCMD - Multiple return paths
16	for $a in input()/article/prolog/ authors/author where empty($a/contact/text()) return <NoContact> {$a/name} </NoContact>	- TCMD - Irregular data
17	for $a in input()/article where contains ($a//p, "the hockey") return <Output> {$a/prolog/title} {$a/body/abstract} </Output>	- TCMD - Text search - One '//' axis
18	for $ent in input()/dictionary/e where $ent/hwg/hw="the" return $ent	- TCSD - Exact match - Reconstruction
19	for $ent in input()/dictionary/e where $ent/*/hw = "and" return $ent/ss/s/qp/*/qt	- TCSD - Path expressions
20	for $a in input()/dictionary/e [hwg/hw="the"]/ss/s/qp/q order by $a/qd return <Output> {$a/a}{$a/qd} </Output>	- TCSD - Sorting
21	for $a in input()/dictionary/e where contains ($a, "hockey") Return $a/hwg/hw	- TCSD - Text search
22	for $ent in input()/dictionary/e where $ent/ss/s/qp/q/qd="1900" return $ent/hwg/hw	- TCSD - Exact match - Deep path

Fig. 3. XBench queries (contd.).



(a) 10 MB (b) 100 MB (c) 1 GB

Fig. 4. Query performance (data-centric).

experiments was 60 minutes; if the execution did not finish in that period, we do not show it in the figures. Note that we use the optimized version of SUCXENT++ for our performance study. In non-optimized SUCXENT++, the join between the **Path** and **PathValue** tables took a significant portion of the query processing time. The optimized version of SUCXENT++ reduces this overhead by altering the translated SQL queries. The join expression `v1.PathId = p1.Id and p1.PathExp =` ***path*** in a translated SQL query is replaced with `v1.PathId =` ***n*** where ***n*** is the `PathId` value corresponding to ***path*** in the table `Path`. Similarly, `v1.PathId = p1.Id and p1.PathExp LIKE` ***path%*** is replaced with `v1.PathId >= n and v1.PathId <= m`. For the second case `PathId`s are assigned in lexicographic order and `(n, m)` correspond to the first and last occurrences of expressions that have the prefix *path*. Furthermore, the optimized version also incorporates strategies to optimize recursive path expressions. The reader may refer to [12, 13] for further details. We now elaborate on the performance results.

Query Q1 [Exact match (DCMD)]: For all three data sizes SHARED-INLINING performs the best. This can be explained based on the relational schema generated for

Shared-Inlining and the SQL query corresponding to query Q1. The Shared-Inlining version involves a single predicate on a single table. All other approaches involve several joins and predicates. Observe that XParent performs marginally better than Sucxent++ for the 10MB data set. This is due to two reasons. First, Sucxent++ involves θ-joins whereas XParent use equi-joins. Second, the data set has several small documents instead of one large document. The translations of both schema-oblivious approaches involve a join of the *DocId* attribute to filter nodes in the same document. This generates much smaller join sizes and the results are particularly obvious for equi-join based queries. However, for the 100MB and 1GB data sets Sucxent++ performs better than XParent by up to 41.7 times. This is due to the following reasons. First, XParent incurs a greater number of joins on a much larger data set. Second, the optimization strategies in Sucxent++ reduce the number of path joins significantly. This reduces the advantage of equi-joins to a great extent as the data size increases.

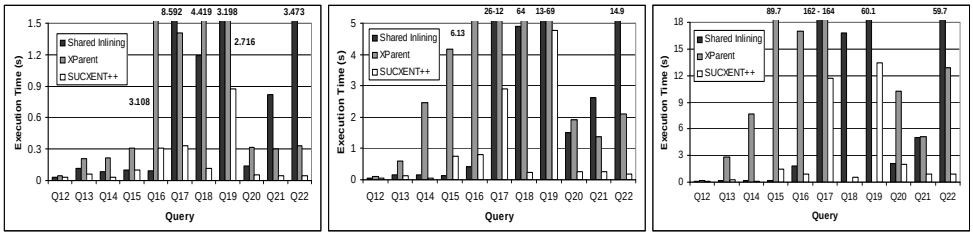
Queries Q2, Q3 [Exact match and ordered access (DCMD)]: Sucxent++ outperforms both Shared-Inlining and XParent for these queries as data size increases (see Figure 6 also). This is because for Shared-Inlining as data size increases the cost of join operation to extract *order_line* and *item_id* (especially with descendant axis) from other tables increases. Sucxent++ performs better than XParent due to its more optimal storage strategy (as discussed above).

Query Q4 [Sort and return multiple elements (DCMD)]: Shared-Inlining performs better than the other two approaches. The main observation here is that Sucxent++ significantly outperforms XParent (up to 970 times) with the increase in data size. This is because the number of joins in the translated SQL statement increases with the number of predicates or return clause elements in the XQuery query. Observe that this query has quite a few return elements (in addition to a sort operation). As a result, XParent requires a greater number of joins (and on larger data) than Sucxent++.

Query Q5 [Document reconstruction (DCMD)]: Shared-Inlining performs better as evaluation of `@id=6` is faster due to smaller (fragmented) tables. But the cost of join becomes higher with increasing data size and therefore the gap between Shared-Inlining and Sucxent++ decreases.

Query Q6 [Join and multiple return elements (DCMD)]: Sucxent++ now performs better than both Shared-Inlining and XParent (up to 1629 times better than XParent). This can be attributed to the fact that even Shared-Inlining has to execute more joins for this query as it involves an XQuery join. XParent is outperformed significantly by other two approaches due to larger number of joins.

Query Q7 [Exact match (DCSD)]: Shared-Inlining performs significantly better for smaller data size than the other two approaches. However, the gap between Sucxent++ and Shared-Inlining decreases with the increase in data size (reasons are similar to the above discussion). Observe that XParent outperforms Sucxent++ for the small data set. However, Sucxent++ performs better than XParent for the 100 MB and 1GB data sets. This is because, unlike the DCMD version of this query, the join on the *DocId* attribute no longer generates small join sizes and the advantages of equi-joins are negated by the large data size in XParent.



(a) 10 MB (b) 100 MB (c) 1 GB

Fig. 5. Query performance (text-centric).

Query Q8 [Quantification (DCSD)]: This query is quite complex as it involves quantification in the form of the `every` clause. The SHARED-INLINING approach performs the best even though it has to execute quite a few joins as well. This is because the execution of the quantification clause is much better in the SHARED-INLINING approach. SUCXENT++ performs significantly better than XParent. In fact, XParent failed to return results in 60 minutes for the 1GB data set and are, therefore, not included in the figures.

Query Q9 [Document Construction (DCSD)]: SHARED-INLINING outperforms SUCXENT++ for 10MB and 100MB data sets. With its inherent greater data fragmentation one would expect SHARED-INLINING to perform the worst. However, the query involves a predicate and the construction is only for a small part of the document - with the entire part available in a single table. As a result SHARED-INLINING performs better. Particularly, for smaller data size the join cost to extract child elements is not significant enough for SHARED-INLINING . Cost of evaluating `@Id=I6` is lower for SHARED-INLINING due to smaller tables. However, as the data size increases the cost of the join increases. As a result, the overall performance difference is not as significant with the increase in data size. In fact, for 1GB data set SUCXENT++ marginally outperforms SHARED-INLINING . Note that the mapping strategies in [15] do not allow mapped documents to be faithfully reconstructed [9]. Hence, the resulting mapping may be lossy.

Query Q10 [Irregular Data (DCSD)]: For SHARED-INLINING , all the data required for this query could be found in just one table. Therefore, SHARED-INLINING performs the best. Also, the empty clause is quite easily implemented by using `= null` in the corresponding SQL statement. The translation for schema-oblivious approaches is quite complicated as there is no notion of `"null"` in these two approaches. Both approaches implement this query using the SQL `not in` clause. As a result the performance is much worse than that of SHARED-INLINING . SUCXENT++ as expected outperforms XParent. In fact, XParent failed to complete execution in 60 minutes for the 100 MB and 1 GB data sets.

Query Q11 [Datatype Casting (DCSD)]: SHARED-INLINING can implement datatype casting in a much cleaner fashion than schema-oblivious approaches. Schema-oblivious approaches (in this case at least) store all data as strings in a single column. SHARED-INLINING uses several different columns and each can be assigned a specific datatype. As a result, the query performance in SHARED-INLINING is much better. In the schema-oblivious approach, data has to be first filtered based on the path

expression and only then can it be typecast. Note that this query has five path expressions. As a result the number of joins in XParent is significantly higher than in SUCXENT++. Therefore, SUCXENT++ significantly outperforms XParent by up to 1700 times.

5 Text-centric Query Processing

Text search plays a very important role in XML document systems. In this section we study the performance of the representative approaches for XML queries on text-centric XML documents. We use the queries Q12 to Q22 in Figure 3 for our performance study. The results are shown in Figure 5. Note that for fair comparison we use the default text processing support of the RDBMS and do not build any additional full text indexes on the underlying RDBMS.

Query Q12 [Exact match (TCMD)]: The performance characteristics for this query are quite similar to those of Q1 except that the difference between SHARED-INLINING and the schema-oblivious approaches is not a significant as in the data-centric case (Q1 to Q3). This is because the query in the SHARED-INLINING approach requires two tables (compared to one in Q1).

Queries [Q13 to Q15 (TCMD)]: For Q13, the evaluation of `@heading="introduction"` is faster for SHARED-INLINING . The join operation to extract child elements is not very expensive for this query. For larger data sizes SHARED-INLINING outperforms SUCXENT++ mainly due to the smaller table that needs to be queried for evaluating `@heading="introduction"`. For SUCXENT++ the **Path-Value** table becomes quite large for larger data sets and hence it takes longer time. Q14 has the `contains` clause which requires all child elements to be queried. This involves a large number of joins for SHARED-INLINING . Consequently, SUCXENT++ performs better. SHARED-INLINING performs better for Q15 for the same reasons as discussed for Q13.

Query Q16 [Irregular Data (TCMD):] Unlike Q10, this query has only two path expressions. The join cost for SHARED-INLINING increases significantly with the increase in data size (especially in the absence of an exact predicate) due to several steps in the path expressions. As a result, even though both the schema-oblivious approaches perform worse than SHARED-INLINING for smaller data size, SUCXENT++ performs better for 1GB data set.

Query Q17 [Text search (TCMD)]: SHARED-INLINING performs significantly worse than the schema-oblivious approaches. This is due to the recursive nature of the query. In SHARED-INLINING , the resolution of the path expression `/article//p` requires the use of the UNION operator on three sub-queries. In addition, one of the sub-queries requires a join across several tables. Due to reasons discussed earlier, SUCXENT++ performs better than XParent.

Query Q18 [Exact match/Reconstruction (TCSD)]: SUCXENT++ again performs the best. Unlike in Q9, SHARED-INLINING has to join four tables to reconstruct the element `e`.

Queries Q19, Q20 [Path expressions (TCSD)]: SUCXENT++ performs better than all the other approaches. XParent performs the worse due to the multiple path expressions in the query. SHARED-INLINING performs slightly worse due to

Query	Shared-Inlining / SUCXENT++			XParent / SUCXENT++		
	10MB	100MB	1GB	10MB	100MB	1GB
Q1	0.89	0.57	0.70	0.98	8.75	41.71
Q2	1.00	0.96	1.59	1.60	27.03	24.60
Q3	0.61	1.04	1.29	1.37	18.21	39.16
Q4	0.47	0.80	0.96	24.58	320.96	970.07
Q5	0.26	0.10	0.15	3.76	8.87	21.78
Q6	1.06	1.55	2.16	26.55	88.91	1629.09
Q7	0.23	0.33	0.92	0.57	1.26	7.06
Q8	0.91	0.77	0.64	10.62	472.97	DNF
Q9	0.76	0.93	1.07	8.29	9.48	16.62
Q10	0.52	0.16	0.12	153.33	DNF	DNF
Q11	0.95	0.33	0.22	84.84	475.41	1699.74

Query	Shared-Inlining / SUCXENT++			XParent / SUCXENT++		
	10MB	100MB	1GB	10MB	100MB	1GB
Q12	1.07	0.67	0.92	1.67	1.81	3.24
Q13	1.96	1.09	0.91	3.62	4.27	11.45
Q14	2.55	2.64	3.85	6.67	45.76	134.26
Q15	1.04	0.17	0.10	3.10	5.56	62.41
Q16	0.30	0.51	1.89	10.02	7.63	18.05
Q17	26.12	9.14	13.90	4.30	4.23	14.06
Q18	10.56	20.63	33.01	39.11	270.90	DNF
Q19	3.67	2.75	4.48	3.12	14.55	DNF
Q20	2.59	5.73	1.04	5.93	7.34	5.07
Q21	16.38	9.89	5.49	6.00	5.21	5.63
Q22	69.46	87.84	68.66	6.70	12.38	14.81

Fig. 6. Performance summary.

multiple joins in the resulting SQL query - although the performance is still comparable to SUCXENT++. Specifically, SHARED-INLINING involves both join and the union operators due to the wild card path expressions `/dictionary/e/*/hw` and `/dictionary/e/ss/s/qp/*/qt`.

Query Q21 [Text search (TCSD)]: SUCXENT++ performs the best for this query again. The query for the SHARED-INLINING approach has several joins in small subqueries that are finally combined with a UNION clause. This is due to the `contains` clause in the XQuery query. In addition, SHARED-INLINING must execute multiple joins to reconstruct the `hw` element. However, with increasing size querying for `"hockey"` in the **PathValue** table becomes more and more expensive compared to searching for it in smaller tables in SHARED-INLINING . Therefore, the gap reduces with increasing data size.

Query Q22 [Exact match (TCSD)]: This query presents a "deep" path expression resulting in several joins for the SHARED-INLINING approach. As a result SUCXENT++ significantly outperforms SHARED-INLINING by up to 88 times.

6 Summary

In this paper, we compared the performance of one schema-conscious approach (SHARED-INLINING [15]) with two representative schema-oblivious approaches (XParent[8] and SUCXENT++[12]). We show that it is indeed possible for a schema-oblivious approach to outperform a schema-conscious approach for several types of *non-recursive* XML queries. Figure 6 provides summary of the performance results for the compared approaches with respect to SUCXENT++. These figures show the ratio of time taken for a given approach to the time taken in SUCXENT++. If a query fails to execute in 60 minutes then we show it as "DNF" in the corresponding column. Observe that SHARED-INLINING outperforms XParent for all queries except Q22. However, SUCXENT++ can be several times faster than the schema-conscious approach (SHARED-INLINING) for text-centric documents; the highest observed factor being 87.8. On the other hand, SHARED-INLINING fares better for large data-centric XML documents as it outperforms SUCXENT++ for 73% of the benchmark queries (Queries Q1 - Q11).

SUCXENT++ significantly outperforms XParent for all non-recursive queries that we have experimented with. In particular, SUCXENT++ is 7-1700 times (excluding

"DNF" queries in XParent) faster than XParent for queries (Q1-Q11) on large data-centric XML documents (1GB). For large text-centric documents, it is still 3-134 times faster. Note that the optimization techniques in SUCXENT++ as described in [12] can also be applied to XParent. A preliminary study of the query plans generated by the RDBMS for XParent suggests that XParent can also benefit from these optimizations. However, a considerable performance difference shall still exists between SUCXENT++ and XParent, especially for large data set, primarily due to XParent's large storage requirements and consequently greater I/O-cost.

References

1. P. BOHANNON, J. FREIRE, P. ROY, J. SIMEON. From XML Schema to Relations: A Cost-based Approach to XML Storage. *In IEEE ICDE*, 2002.
2. T. BÖHME, E. RAHM. XMach-1: A Benchmark for XML Data Management. *In German Database Conference*, 2001.
3. S. BRESSAN, M-L. LEE, Y. G. LI, Z. LACROIX, U. NAMBIAR. The XOO7 Benchmark.*In EEXTT*, 2002.
4. M. CAREY, D. DEWITT, J. NAUGHTON. The OO7 Benchmark. *In ACM SIGMOD*, 1993.
5. D. DEHAAN, D. TOMAN, M. P. CONSENS, M. T. OZSU. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Coding. *In ACM SIGMOD*, 2003.
6. L. ENNSER, C. DELPORTE, M. OBA, K. SUNIL. Integrating XML and DB2 XML Extender and DB2 Text Extender. *IBM Redbooks*, 2001.
7. D. FLORESCU, D. KOSSMAN. Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin.* 22(3), 1999.
8. H. JIANG, H. LU, W. WANG AND J. XU YU. Path Materialization Revisited: An Efficient Storage Model for XML Data. *13th Australasian Database Conference (ADC'02)* , 2002.
9. R. KRISHNAMURTHY, R. KAUSHIK, J. F. NAUGHTON. XML to SQL Query Translation Literature: The State of the Art and Open Problem.*In XSym*, 2003.
10. R. KRISHNAMURTHY, V. T. CHAKARAVARTHY, R. KAUSHIK, J. F. NAUGHTON. Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation.*In IEEE ICDE*, 2004.
11. H. LU, H. JIANG, J. X. XU, G. YU ET AL. What Makes the Differences: Benchmarking XML Database Implementations. *In ACM Trans. on Internet Technology*, 5(1), 2005.
12. S. PRAKASH, S. S. BHOWMICK, S. K. MADRIA. Efficient Recursive XML Query Processing Using Relational Databases. *In ER*, 2004.
13. S. PRAKASH, S. S. BHOWMICK, S. K. MADRIA. Efficient Recursive XML Query Processing Using Relational Databases. *To appear in Data and Knowledge Engineering Journal*, Special Issue on Best Papers of ER 2004, Elsevier Science, 2006.
14. A. SCHMIDT, F. WAAS, M. KERSTEN, M. J. CAREY, I. MANOLESCU AND R. BUSSE. XMark: A Benchmark for XML Data Management. *In VLDB*, 2002.
15. J. SHANMUGASUNDARAM, K. TUFTE ET AL. Relational Databases for Querying XML Documents: Limitations and Opportunities. *In VLDB 1999.*
16. F. TIAN, D. DEWITT, J. CHEN AND C. ZHANG. The Design and Performance Evaluation of Alternative XML Storage Strategies. *ACM Sigmod Record, Vol. 31(1), 2002.*
17. B. YAO, M. TAMER ÖZSU, N. KHANDELWAL. XBench: Benchmark and Performance Testing of XML DBMSs. *In ICDE*, Boston, 2004.
18. M. YOSHIKAWA, T. AMAGASA, T. SHIMURA, AND S. UEMURA. XRel: A Path-based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM TOIT* 1(1):110-141, 2001.
19. C. ZHANG, J. NAUGHTON, D. DEWITT, Q. LUO AND G. LOHMANN. On Supporting Containment Queries in Relational Database Systems. *In ACM SIGMOD*, 2001.
20. Microsoft SQL Server 2000 SDK Documentation, Microsoft 2000, `http://www.microsoft.com`.
21. Oracle XML DB. `http://www.oracle.com`.