The Michigan Benchmark: Towards XML Query Performance Diagnostics

Kanda Runapongsa Jignesh M. Patel H. V. Jagadish Shurug Al-Khalifa
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109, USA
{krunapon, jignesh, jag, shurug}@eecs.umich.edu

Abstract

We propose a *micro-benchmark* for XML data management to aid engineers in designing improved XML processing engines. This benchmark is inherently different from application-level benchmarks, which are designed to help users choose between alternative products. We primarily attempt to capture the rich variety of data structures and distributions possible in XML, and to isolate their effects, without imitating any particular application. The benchmark specifies a single data set against which carefully specified queries can be used to evaluate system performance for XML data with various characteristics.

We have used the benchmark to analyze the performance of two database systems: a native XML DBMS and a commercial ORDBMS. The benchmark reveals key strengths and weaknesses of these systems. We find that commercial relational techniques are effective for XML query processing in many cases, but are sensitive to query rewriting, and require better support for efficiently determining indirect structural containment.

1 Introduction

XML query processing has taken on considerable importance recently, and several XML databases [3, 8–10, 12, 24, 26] have been constructed on a variety of platforms. There has naturally been an interest in benchmarking the performance of these systems, and a number of benchmarks have been proposed [6, 19, 21]. The focus of currently proposed benchmarks is to assess the performance of a given XML database in performing a variety of representative tasks. Such benchmarks are valuable to potential users of a database system in providing an indication of the performance that the user can expect on their specific application. The challenge is to devise benchmarks that are sufficiently representative of the requirements of "most" users. The TPC series of benchmarks accomplished this, with reasonable success, for relational database systems. However, no benchmark has been successful in the realm of ORDBMS and OODBMS which have extensibility and user defined functions that lead to great heterogeneity in the nature of their use. It is too soon to say whether any of the current XML benchmarks will be successful in this respect - we certainly hope that they will.

One aspect that current XML benchmarks do not focus on is the performance of the basic query evaluation operations such as selections, joins, and aggregations. A "micro-benchmark" that highlights the performance of these basic operations can be very helpful to a database developer in understanding and evaluating alternatives for implementing these basic operations. A number of questions related to performance may need to be answered: What are the strengths and weaknesses of specific access methods? Which areas should the developer focus attention on? What is the basis to choose between two alternative implementations? Questions of this nature are central to well-engineered systems. Application-level benchmarks, by their nature, are unable to deal with these important issues in detail. For relational systems, the Wisconsin benchmark [11] provided the database community with an invaluable engineering tool to assess the performance of individual operators and access methods. The work presented in this paper is inspired by the simplicity and the effectiveness of the Wisconsin benchmark for measuring and understanding the performance of relational DBMSs. The goal of this paper is to develop a comparable benchmarking tool for XML data management systems. The benchmark that we propose to achieve this goal is called the Michigan benchmark.

A challenging issue in designing any benchmark is the choice of the benchmark's data set. If the data is specified to represent a particular "real application", it is likely to be quite uncharacteristic for other applications with different data characteristics. Thus, holistic benchmarks can succeed only if they are able to find a real application with data characteristics that are reasonably representative for a large class of different applications.

For a micro-benchmark, the challenges are different. The benchmark data set must be *complex* enough to incorporate data characteristics that are likely to have an impact on the performance of query operations. However, at the same time, the benchmark data set must be *simple* so that it is not only easy to pose and understand queries against the data set, but also easy to pinpoint the component of the system that is performing poorly. We attempt to achieve this balance by using a data set that has a simple schema but carefully orchestrated structure. In addition, random number generators are used sparingly in generating the benchmark's data set. The Michigan benchmark uses random generators for only two attribute values, and derives all other data parameters from these two generated values. Furthermore, as in the Wisconsin benchmark, we use appropriate attribute names to reflect the domain and distribution of the attribute values.

When designing benchmark data sets for relational systems, the primary data characteristics that are of interest are the distribution and domain of the attribute values and the cardinality of the relations. Moreover, there may be a few additional secondary characteristics, such as clustering and tuple/attribute size. In XML databases, besides the distribution and domain of attribute values and cardinality, there are several other characteristics, such as tree fanout and tree depth, that are related to the structure of XML documents and contribute to the rich structure of XML data. An XML benchmark must incorporate these additional features into the benchmark data and query set design. The Michigan benchmark achieves this by using a data set that incorporates these characteristics without introducing unnecessary complexity into the data set generation, and by carefully designing the benchmark queries that test the impact of these characteristics on individual query operations.

The main contributions of this paper are:

- The identification of XML data characteristics that may impact the performance of XML query processing engines.
- A single heterogeneous data set against which carefully specified queries can be used to evaluate system performance for XML data with various characteristics.
- Insights from running this benchmark on two database systems: a native XML database system and a commercial object-relational database management system (ORDBMS).

The remainder of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we present the rationale of the benchmark data set design. In Section 4, we describe the queries of the benchmark data set. In Section 5, we present the results of experimental evaluation of DBMSs using the benchmark. We conclude with some final remarks in Section 6.

2 Related Work

Several proposals for generating synthetic XML data have been proposed [1, 5]. Aboulnaga et al. [1] proposed a data generator that accepts as many as 20 parameters to allow a user to control the properties of the generated data. Such a large number of parameters adds a level of complexity that may interfere with the ease of use of a data generator. Furthermore, this data generator does not make available the schema of the data which some systems could exploit. Most recently, Barbosa et al. [5] proposed a template-based data generator for XML, which can generate multiple tunable data sets. In contrast to these previous data generators, the data generator in this proposed benchmark produces an XML data set designed to test different XML data characteristics that may affect the performance of XML engines. In addition, the data generator requires only few parameters to vary the scalability of the data set. The schema of the data set is also available to exploit.

Three benchmarks have been proposed for evaluating the performance of XML data management systems [6, 19, 21]. XMach-1 [6] and XMark [21] generate XML data that models data from particular Internet applications. In XMach-1 [6], the data is based on a web application that consists of text documents, schema-less data, and structured data. In XMark [21], the data is based on an Internet auction application that consists of relatively structured and data-oriented parts. XOO7 [19] is an XML version of the OO7 Benchmark [17], which is a benchmark for OODBMSs. The OO7 schema and instances are mapped into a Document Type Definition (DTD), and the eight OO7 queries are translated into three respective languages for query processing engines: Lore [14, 18], Kweelt [20], and an ORDBMS. While each of these benchmarks provides an excellent measure of how a test system would perform against data and queries in their targeted XML application, it is difficult to extrapolate the results to data sets and queries that are different from ones in the targeted domain. Although the queries in these benchmarks are designed to test different performance aspects of XML engines, they cannot be used to perceive the system performance change as the XML data characteristics change. On the other hand, we have different queries to analyze the system performance with respect to different XML data characteristics, such as tree fanout and tree depth; and different query characteristics, such as predicate selectivity.

Finally, we note that [2] presents desiderata for an XML database benchmark, identifies key components and operations, and enumerates ten challenges that the XML benchmark should address. The central focus of this work is application-level benchmarks, rather than micro-benchmarks of the sort we propose.

3 Benchmark Data Set

In this section, we first discuss the characteristics of XML data sets that can have a significant impact on the performance of query operations. Then, we present the schema and the generation algorithm for the benchmark data.

3.1 A Discussion of the Data Characteristics

In a relational paradigm, the primary data characteristics are the selectivity of attributes (important for simple selection operations) and the join selectivity (important for join operations). In an XML paradigm, there are several complicating characteristics to consider as discussed in Section 3.1.1 and Section 3.1.2.

3.1.1 Depth and Fanout

Depth and fanout are two structural parameters important to tree-structured data. The depth of the data tree can have a significant performance impact, for instance, when computing indirect containment relationships between ancestor and descendant nodes in the tree. Similarly, the fanout of the tree node can affect the way in which the DBMS stores the data, and answers queries that are based on selecting children in a specific order (for example, selecting the last child).

One potential way of evaluating the impact of fanout and depth is to generate a number of distinct data sets with different values for each of these parameters and then run queries against each data set. The drawback of this approach is that the large number of data sets makes the benchmark harder to run and understand. Instead, our approach is to fold these into a single data set.

Level	Fanout	Nodes	% of Nodes
1	2	1	0.0
2	2	2	0.0
3	2	4	0.0
4	2	8	0.0
5	13	16	0.0
6	13	208	0.0
7	13	2,704	0.4
8	1/13	35,152	4.8
9	2	2,704	0.4
10	2	5,408	0.7
11	2	10,816	1.5
12	2	21,632	3.0
13	2	43,264	6.0
14	2	86,528	11.9
15	2	173,056	23.8
16	–	346,112	47.6

Figure 1: Distribution of the Nodes in the Base Data Set

We create a base benchmark data set of a depth of 16. Then, using a "level" attribute of an element, we can restrict the scope of the query to data sets of certain depth, thereby, quantifying the impact of the depth of the data tree. Similarly, we specify high (13) and low (2) fanouts at different levels of the tree as shown in Figure 1. The fanout of 1/13 at level 8 means that every thirteenth node at this level has a single child, and all other nodes are childless leaves. This variation in fanout is designed to permit queries that focus isolating the fanout factor. For instance, the number of nodes is the same (2,704) at levels 7 and 9. Nodes at level 7 have a fanout of 13, whereas nodes at level 9 have a fanout of 2. A pair of queries, one against each of these two levels, can be used to isolate the impact of fanout.

3.1.2 Data Set Granularity

To keep the benchmark simple, we choose a single large document tree as the default data set. If it is important to understand the effect of document granularity, one can modify the benchmark data set to treat each node at a given level as the root of a distinct document. One can compare the performance of queries on this modified data set against queries on the original data set.

A good benchmark needs to be able to scale in order to measure the performance of databases on a variety of platforms. In the relational model, scaling a benchmark data set is easy - we simply increase the number of tuples. However, with XML, there are many scaling options, such as increasing number of nodes, depth, or fanout. We would like to isolate the effect of the number of nodes from the effects due to other structural changes, such as depth and fanout. We achieve this by keeping the tree depth constant for all scaled versions of the data set and changing the number of fanouts of nodes at only a few levels, namely levels 5–8. In the design of the benchmark data sets, we deliberately keep the fanout of the bottom few levels of the tree constant. This design implies that the percentage of nodes in the lower levels of the tree (levels 9–16) is nearly constant across all the scaled data sets. This allows us to easily express queries that focus on a specified percentage of the total number of nodes in the database. For example, to select approximately 1/16 of all the nodes, irrespective of the scale factor, we use the predicate aLevel = 13.

Due to space limitation, more details regarding the scaling of the benchmark data set are suppressed here. An interested reader can find more details at [25].

3.2 Schema of Benchmark Data

The construction of the benchmark data is centered around the element type BaseType. Each BaseType element has the following attributes:

1. aUnique1: A unique integer generated by traversing the entire data tree in a breadth-first manner. This attribute also serves as the element identifier.
2. aUnique2: A unique integer generated randomly.
3. aLevel: An integer set to store the level of the node.
4. aFour: An integer set to aUnique2 mod 4.
5. aSixteen: An integer set to aUnique1 + aUnique2 mod 16. This attribute is generated using *both* the unique attributes to avoid a correlation between the value of this attribute and other *derived* attributes.
6. aSixtyFour: An integer set to aUnique2 mod 64.
7. aString: A string approximately 32 bytes in length.

The content of each BaseType element is a long string that is approximately 512 bytes in length. The generation of the element content and the string attribute aString is described in Section 3.3.

In addition to the attributes listed above, each BaseType element has two sets of subelements. The first is of type BaseType. The number of repetitions of this subelement is determined by the fanout of the parent element, as described in Figure 1. The second subelement is an OccasionalType, and can occur either 0 or 1 time. The presence of the OccasionalType element is determined by the value of the attribute aSixtyFour of the parent element. A BaseType element has a nested (leaf) element of type OccasionalType if the aSixtyFour attribute has the value 0. An OccasionalType element has content that is identical to the content of the parent but has only one attribute, aRef. The OccasionalType element refers to the BaseType node with aUnique1 value equal to the parent's aUnique1-11 (the reference is achieved by assigning this value to aRef attribute.) In the case where there is no BaseType element has the parent's aUnique1-11 value (e.g., top few nodes in the tree), the OccasionalType element refers to the root node of the tree. The XML Schema specification of the benchmark data is shown in Figure 2.

3.3 Generating the String Attributes and Element Content

The element content of each BaseType element is a long string. Since this string is meant to simulate a piece of text in a natural language, it is not appropriate to generate this string from a uniform distribution. Selecting pieces of text from real sources, however, involves many difficulties, such as how to maintain roughly constant size for each string, how to avoid idiosyncrasies associated with the specific source, and how to generate more strings as required for a scaled benchmark. Moreover, we would like to have benchmark results applicable to a wide variety of languages and domain vocabularies.

To obtain string values that have a distribution similar to the distribution of a natural language text, we generate these long strings synthetically, in a carefully stylized manner. We begin by creating a pool of $2^{16}-1$ (over sixty thousands) [1] synthetic words. The words are divided into 16 buckets, with exponentially growing bucket occupancy. Bucket i has 2^{i-1} words. For example, the first bucket has only one word, the second has two words, the third has four words, and so on. Each made-up word contains information about the bucket from which it is drawn and the word number in the bucket. For example, "15twentynineB14" indicates that this is the 1,529th word from the fourteenth bucket. To keep the size of the vocabulary in the last bucket at roughly 30,000 words, words in the last bucket are derived from words in the other buckets by adding the suffix "ing" (to get exactly 2^{15} words in the sixteenth bucket, we add the dummy word "oneB0ing").

The value of the long string is generated from the template shown in Figure 3, where "PickWord" is actually a placeholder for a word picked from the word pool described above. To pick a word for "PickWord", a bucket is chosen, with each bucket equally likely, and then a word is picked from the chosen bucket, with each word equally likely. Thus, we obtain a discrete Zipf distribution of parameter roughly 1. We use the Zipf distribution since it seems to reflect word occurrence probabilities accurately in a wide variety of situations. The value of aString attribute is simply the first line of the long string that is stored as the element content.

[1] Roughly twice the number of entries in the second edition of the Oxford English Dictionary. However, half the words that are used in the benchmark are "derived" words, produced by appending "ing" to the end of a word.

```
<?xml version="1.0"?>
  <xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.eecs.umich.edu/db/mbench/bm.xsd"
    xmlns="http://www.eecs.umich.edu/db/mbench/bm.xsd"
    elementFormDefault="qualified">
  <xsd:element name="eNest" type="BaseType">
  <xsd:complexType name="BaseType" mixed="true">
  <xsd:sequence>
    <xsd:element name="eNest" type="BaseType" maxOccurs="unbounded">
      <xsd:key name="aU1PK">
        <xsd:selector xpath=".//eNest"/>
        <xsd:field xpath="@aUnique1"/>
      </xsd:key>
      <xsd:unique name="aU2">
        <xsd:selector xpath=".//eNest"/>
        <xsd:field xpath="@aUnique2"/>
      </xsd:unique>
    </xsd:element>
    <xsd:element name="eOccasional" type="OccasionalType" minOccurs="0">
      <xsd:keyref name="aU1FK" refer="aU1PK">
        <xsd:selector xpath="../eOccasional"/>
        <xsd:field xpath="@aRef"/>
      </xsd:keyref>
    </xsd:element>
  </xsd:sequence>
  <xsd:attributeGroup ref="BaseTypeAttrs"/>
  </xsd:complexType>
  <xsd:complexType name="OccasionalType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="aRef" type="xsd:integer" use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:attributeGroup name="BaseTypeAttrs">
    <xsd:attribute name="aUnique1" type="xsd:integer" use="required"/>
    <xsd:attribute name="aUnique2" type="xsd:integer" use="required"/>
    <xsd:attribute name="aLevel" type="xsd:integer" use="required"/>
    <xsd:attribute name="aFour" type="xsd:integer" use="required"/>
    <xsd:attribute name="aSixteen" type="xsd:integer" use="required"/>
    <xsd:attribute name="aSixtyFour" type="xsd:integer" use="required"/>
    <xsd:attribute name="aString" type="xsd:string" use="required"/>
  </xsd:attributeGroup>
</xsd:schema>
```

Figure 2: Benchmark Specification in XML Schema

Sing a song of PickWord,
A pocket full of PickWord
Four and twenty PickWord
All baked in a PickWord.

When the PickWord was opened,
The PickWord began to sing;
Wasn't that a dainty PickWord
To set before the PickWord?

The King was in his PickWord,
Counting out his PickWord;
The Queen was in the PickWord
Eating bread and PickWord.

The maid was in the PickWord
Hanging out the PickWord;
When down came a PickWord,
And snipped off her PickWord!

Figure 3: Generation of the String Element Content

Through the above procedures, we now have the data set that has the structure that facilitates the study of the impact of data characteristics on system performance, and the element/attribute content that simulates a piece of text in a natural language.

4 Benchmark Queries

In creating the data set above, we make it possible to tease apart data with different characteristics, and to issue queries with well-controlled yet vastly differing data access patterns. We are more interested in evaluating the cost of individual pieces of core query functionality than in evaluating the composite performance of queries that are of application-level. Knowing the costs of individual basic operations, we can estimate the cost of any complex query by just adding up relevant piecewise costs (keeping in mind the pipelined nature of evaluation, and the changes in sizes of intermediate results when operators are pipelined).

One clean way to decompose complex queries is by means of an algebra. While the benchmark is not tied to any particular algebra, we find it useful to refer to queries as "selection queries", "join queries" and the like, to clearly indicate the functionality of each query. A complex query that involves many of these simple operations can take time that varies monotonically with the time required for these simple components.

In the following subsections, we describe each of these different types of queries in detail. In these queries, the types of the nodes are assumed to be `BaseType` (`eNest` nodes) unless specified otherwise.

4.1 Selection

Relational selection identifies the tuples that satisfy a given predicate over its attributes. XML selection is both more complex and more important because of the tree structure. Consider a query, against a popular bibliographic database, that seeks books, published in the year 2002, by an author with name including the string "Blake". This apparently straightforward selection query involves matches in the database to a 4-node "query pattern", with predicates associated with each of these four (namely book, year, author, and name). Once a match has been found for this pattern, we may be interested in returning only the book element, all the nodes that participated in the match, or various other possibilities. We attempt to organize the various sources of complexity in the following.

4.1.1 Returned Structure

In a relation, once a tuple is selected, the tuple is returned. In XML, as we saw in the example above, once an element is selected, one may return the element, as well as some structure related to the element, such as the sub-tree rooted at the element. Query performance can be significantly affected by how the data is stored and when the returned result is materialized.

To understand the role of returned structure in query performance, we use the query, "Select all elements with aSixtyFour = 2." The selectivity of this query is 1/64 (1.6%)[2].

This query is run in the following cases:

- **QR1.** Return only the elements in question, not including any subelements.
- **QR2.** Return the elements and all their immediate children.
- **QR3.** Return the entire sub-tree rooted at the elements.
- **QR4.** Return the elements and their selected descendants with aFour = 1.

The remaining queries in the benchmark simply return the unique identifier attributes of the selected nodes (aUnique1 for eNest and aRef for eOccasional), except when explicitly specified otherwise. This design choice ensures that the cost of producing the final result is a small portion of the query execution cost.

4.1.2 Simple Selection

Even XML queries involving only one element and few predicates can show considerable diversity. We examine the effect of this simple selection predicate in this set of queries.

- **Exact Match Attribute Value Selection**

  Selection based on the value of a string attribute.

  QS1. Low selectivity. Select nodes with aString = "Sing a song of oneB4". Selectivity is 0.8%.

  QS2. High selectivity. Select nodes with aString = "Sing a song of oneB1". Selectivity is 6.3%.

[2]Note that details about the computation of the selectivities of queries can be found at [25].

Selection based on the value of an integer attribute.

These following queries have almost the same selectivities as the above string attribute queries.

QS3. Low selectivity. Select nodes with aLevel = 10. Selectivity is 0.7%.

QS4. High selectivity. Select nodes with aLevel = 13. Selectivity is 6.0%.

Selection on range values.

QS5. Select nodes with aSixtyFour between 5 and 8. Selectivity is 6.3%.

Selection with sorting.

QS6. Select nodes with aLevel = 13 and have the returned nodes sorted by aSixtyFour attribute. Selectivity is 6.0%.

Multiple-attribute selection.

QS7. Select nodes with attributes aSixteen = 1 and aFour = 1. Selectivity is 1.6%.

- **Element Name Selection**

  QS8. Select nodes with the element name eOccasional. Selectivity is 1.6%.

- **Order-based Selection**

  QS9. High fanout. Select the second child of every node with aLevel = 7. Selectivity is 0.4%.

  QS10. Low fanout. Select the second child of every node with aLevel = 9. Selectivity is 0.4%.

  Since the fraction of nodes in these two queries are the same, the performance difference between them is likely to be on account of fanout.

- **Element Content Selection**

  QS11. Low selectivity. Select OccasionalType nodes that have "oneB4" in the element content. Selectivity is 0.2%.

  QS12. High selectivity. Select nodes that have "oneB4" as a substring of element content. Selectivity is 12.5%.

- **String Distance Selection**

  QS13. Low selectivity. Select all nodes with element content that the distance between keyword "oneB5" and keyword "twenty" is not more than four. Selectivity is 0.8%.

  QS14. High selectivity. select all nodes with element content that the distance between keyword "oneB2" and keyword "twenty" is not more than four. Selectivity is 6.3%.

4.1.3 Structural Selection

Selection in XML is often based on patterns. Queries should be constructed to consider multi-node patterns of various sorts and selectivities. These patterns often have "conditional selectivity." Consider a simple two node selection pattern. Given that one of the nodes has been identified, the selectivity of the second node in the pattern can differ from its selectivity in the database as a whole. Similar dependencies between different attributes in a relation could exist, thereby affecting the selectivity of a multi-attribute predicate. Conditional selectivity is complicated in XML because different attributes may not be in the same element, but rather in different elements that are structurally related.

All queries listed in this section return only the root of the selection pattern, unless specified otherwise. In these queries, the selectivity of a predicate is noted following the predicate.

- **Order-Sensitive Parent-Child Selection**

  QS15. Local ordering. Select the second element below *each* element with aFour = 1 (sel=1/4) if that second element also has aFour = 1 (sel=1/4). Selectivity is 3.1%.

  QS16. Global ordering. Select the second element with aFour = 1 (sel=1/4) below *any* element with aSixtyFour = 1 (sel=1/64). This query returns at most one element, whereas the previous query returns one for each parent.

  QS17. Reverse ordering. Among the children with aSixteen = 1 (sel=1/16) of the parent element with aLevel = 13 (sel=6.0%), select the last child. Selectivity is 0.7%.

- **Parent-Child Selection**

  QS18. Medium selectivity of both parent and child. Select nodes with aLevel = 13 (sel=6.0%, approx. 1/16) that have a child with aSixteen = 3 (sel=1/16). Selectivity is approximately 0.7%.

  QS19. High selectivity of parent and low selectivity of child. Select nodes with aLevel = 15 (sel=23.8%, approx. 1/4) that have a child with aSixtyFour = 3 (sel=1/64). Selectivity is approximately 0.7%.

  QS20. Low selectivity of parent and high selectivity of child. Select nodes with aLevel = 11 (sel=1.5%, approx. 1/64) that have a child with aFour = 3 (sel=1/4). Selectivity is approximately 0.7%.

- **Ancestor-Descendant Selection**

  QS21. Medium selectivity of both ancestor and descendant. Select nodes with aLevel = 13 (sel=6.0%, approx. 1/16) that have a descendant with aSixteen = 3 (sel=1/16). Selectivity is 3.5%.

  QS22. High selectivity of ancestor and low selectivity of descendant. Select nodes with aLevel = 15 (sel=23.8%, approx. 1/4) that have a descendant with aSixtyFour = 3 (sel=1/64). Selectivity is 0.7%.

  QS23. Low selectivity of ancestor and high selectivity of descendant. Select nodes with aLevel = 11 (sel=1.5%, approx. 1/64) that have a descendant with aFour = 3 (sel=1/4). Selectivity is 1.5%.

- **Ancestor Nesting in Ancestor-Descendant Selection**

  In the ancestor-descendant queries above (QS21-QS23), ancestors are never nested below other ancestors. To test the performance of queries when ancestors are recursively nested below other ancestors, we have three other ancestor-descendant queries. These queries are variants of QS21-QS23.

  QS24. Medium selectivity of both ancestor and descendant. Select nodes with aSixteen = 3 (sel=1/16) that have a descendant with aSixteen = 5 (sel=1/16).

QS25. High selectivity of ancestor and low selectivity of descendant. Select nodes with aFour = 3 (sel=1/4) that have a descendant with aSixtyFour = 3 (sel=1/64).

QS26. Low selectivity of ancestor and high selectivity of descendant. Select nodes with aSixtyFour = 9 (sel=1/64) that have a descendant with aFour = 3 (sel=1/4).

The overall selectivities of these queries (QS24-QS26) cannot be the same as that of the "equivalent" unnested queries (QS21-QS23) for two situations – first, the same descendants can now have multiple ancestors they match, and second, the number of candidate descendants is different (fewer) since the ancestor predicate can be satisfied by nodes at any level (and will predominantly be satisfied by nodes at levels 15 and 16, due to their large numbers). These two effects may not necessary cancel each other out. We focus on the local predicate selectivities and keep these the same for all of these queries (as well as for the parent-child queries considered before).

QS27. Similar to query QS26, but return both the root node and the descendant node of the selection pattern. Thus, the returned structure is a pair of nodes with an inclusion relationship between them.

- **Complex Pattern Selection**

Complex pattern matches are common in XML databases, and in this section, we introduce a number of *chain* and *twig* queries that we use in this benchmark. Figure 4 shows an example for these query types. In the figure, each node represents a predicate such as an element tag name predicate, or an attribute value predicate, or an element content match predicate. A structural parent-child relationship in the query is shown by a single line, and an ancestor-descendant relationship is represented by a double-edged line. The chain query shown in the Figure 4(i) finds all nodes matching condition A, such that there is a child matching condition B, such that there is a child matching condition C, such that there is a child matching condition D. The twig query shown in the Figure 4(ii) matches all nodes that satisfy condition A, and have a child node that satisfies condition B, and also have a descendant node that satisfies condition C.

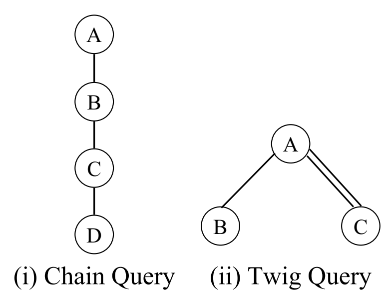

Figure 4: Samples of Chain and Twig Queries

We use the following complex queries in our benchmark:

- **Parent-Child Complex Pattern Selection**

QS28. One chain query with three parent-child joins with the selectivity pattern: high-low-low-high. The query is to test the choice of join order in evaluating a complex query. To achieve the desired selectivities, we use the following predicates: aFour=3 (sel=1/4), aSixteen=3 (sel=1/16), aSixteen=5 (sel=1/16) and aLevel=16 (sel=47.6%).

QS29. One twig query with two parent-child joins with the selectivity pattern: low-high, low-low. Select parent nodes with aLevel = 11 (sel=1.5%) that have a child with aFour = 3 (sel=1/4), and another child with aSixtyFour = 3 (sel=1/64).

QS30. One twig query with two parent-child joins with the selectivity pattern: high-low, high-low. Select parent nodes with aFour = 1 (sel=1/4) that have a child with aLevel = 11 (sel=1.5%) and another child with aSixtyFour = 3 (sel=1/64).

- **Ancestor-Descendant Complex Pattern Selection**

  QS31-QS33. Repeat queries QS28-QS30, but using ancestor-descendant in place of parent-child.

  QS34. One twig query with one parent-child join and one ancestor-descendant join. Select nodes with aFour = 1 (sel=1/4) that have a child of nodes with aLevel = 11 (sel=1.5%) and a descendant with aSixtyFour = 3 (sel=1/64).

- **Negated Selection**

  QS35. Find all BaseType elements below which there is no OccasionalType element.

4.2 Value-Based Join

A value-based join involves comparing values at two different nodes that need not be related structurally. In computing the value-based joins, one would naturally expect *both* nodes participating in the join to be returned. As such, the return structure is the pair of the aUnique1 attributes of nodes joined.

QJ1. Low selectivity. Select nodes with aSixtyFour = 2 (sel=1/64) and join with themselves based on the equality of aUnique1 attribute. The selectivity of this query is approximately 1.6%.

QJ2. High selectivity. Select nodes based on aSixteen = 2 (sel=1/16) and join with themselves based on the equality of aUnique1 attribute. The selectivity of this query is approximately 6.3%.

4.3 Pointer-Based Join

These queries specify joins using references that are specified in the DTD or XML Schema, and the implementation of references may be optimized with logical OIDs in some XML databases.

QJ3. Low selectivity. Select all OccasionalType nodes that point to a node with aSixtyFour = 3 (sel=1/64). Selectivity is 0.02%.

QJ4. High selectivity. Select all OccasionalType nodes that point to a node with aFour = 3 (sel=1/4). Selectivity is 0.4%.

Both of these pointer-based joins are semi-join queries. The returned elements are only the eOccasional nodes, not the nodes pointed to.

4.4 Aggregation

Aggregate queries are very important for data warehousing applications. In XML, aggregation also has richer possibilities due to the structure. These are explored in the next set of queries.

QA1. Value aggregation. Compute the average value of the aSixtyFour attribute of all nodes at level 15 (have aLevel = 15 (sel=23.8%)). The number of returned nodes is 1.

QA2. Value aggregation with groupby. Compute the average value of the aSixtyFour attribute of all nodes at each level. The return structure is a tree, with a dummy root and a child for each group. Each leaf (child) node has one attribute for the level and one attribute for the average value. The number of returned trees is 16.

QA3. Value aggregate selection. Select elements that have at least two occurrences of keyword "oneB1" (sel=1/16) in their content. Selectivity is 0.3%.

QA4. Structural aggregation. Amongst the nodes at level 11 (have aLevel = 11 (sel=1.5%)), find the node(s) with the largest fanout. Selectivity is 0.02%.

QA5. Structural aggregate selection. Select elements that have at least two children that satisfy aFour = 1 (sel=1/4). Selectivity is 3.1%.

QA6. Structural exploration. For each node at level 7 (have aLevel = 7 (sel=0.4%)), determine the height of the sub-tree rooted at this node. Selectivity is 0.4%.

There are also other functionalities, such as casting, which can be significant performance factors for engines that need to convert data types. However, in this benchmark, we focus on testing the core functionality of the XML engines.

4.5 Update

The benchmark also contains seven update queries, which include point update and delete, bulk insert and delete, bulk load, bulk reconstruction, and bulk restructuring. In the interest of space, we omit the update queries in this paper and refer an interested reader to [25].

5 The Benchmark in Action

In this section, we present and analyze the performance of different databases using the Michigan benchmark. We conducted experiments using one native XML database, and one leading commercial ORDBMS. The native XML database is TIMBER [26], and for the ORDBMS we used the Hybrid inlining algorithm to map the data into a relational schema [23]. To generate good SQL queries, we adopted the algorithm presented in [13]. The SQL queries used can be found at [25].

We would have preferred to report results on a commercial XML database, but were not successful in loading the entire base data set into any of three products that we tried. We are currently working with the vendor of one of these products to load and benchmark an entire data set.

5.1 Experimental Platform and Methodology

All experiments were run on a single-processor 550 MHz Pentium III machine with 256 MB of main memory. The benchmark machine was running the Windows operating system and was configured with a 20 GB Western Digital WDC WD200BB-00AUA1 IDE disk. Both TIMBER and the ORDBMS use a 64 MB buffer pool size. For the ORDBMS, we used the default settings that the ORDBMS chooses during the software installation. The only setting that we changed was to enable the use of hash joins, as we found that query response times generally improved with this option turned on. After loading the data we update all statistics to provide the optimizer with the most current statistical information.

For this experiment we loaded the base data set (739K nodes, about 500MB of raw data). Although we wanted to load larger scaled up data sets, we found that in many cases the parsers are fragile and break down with large documents. Consequently, for this study, we decided to load another *scaled down* version of the data. The scaled down data set, which we call **ds0.1x**, is produced by changing the fanouts of the nodes at levels 5, 6, 7 and 8 to 4, 4, 4 and 1/4 respectively. This scaled down data set is approximately $1/10$th the size of the **ds1x** data set. Note that because of the nature of the document tree, the percentage of nodes at the levels close to the leaves remains the same, hence the query selectivities stay roughly constant even in this scaled down data set.

For the purpose of the experiment, we loaded and wrote queries against the scaled down set before using the base data set. The smaller data set size reduced the time to set up the queries and load the scripts, for both TIMBER and the commercial ORDBMS. The same queries and scripts could be reused for the base data set. Since we expect that this strategy may also be useful to other users of this benchmark, the data generator for this benchmark, which is available for free download from the benchmark's website [25], allows generation of this scaled down data set.

In our experiments, each query was executed five times, and the execution times reported in this section is an average of the middle three runs, excluding the fastest and the slowest runs. Queries were always run in "cold" mode, so the query execution times do not include side-effects of cached buffer pages from a previous run of another query or a previous run of the same query.

In our own use of the benchmark, we have found it useful to produce two kinds of tables: a *summary* table which presents a single number for a group of related queries, and a *detail* table that shows the query execution time for each individual query. The summary table presents a high-level view of the performance of the benchmark queries. It contains one entry for a *group* of related queries, and shows the geometric mean of the response times of the queries in that group. Figure 5 shows the summary table for the systems that we benchmarked. In the subsequent sections, we present the individual query execution times, and analyze the queries in each; Figure 5 also indicates the sections in which the detailed numbers are presented and analyzed. In the figures *N/A* indicates that queries could not be run with the given configuration and system software.

From Figure 5, we observe that TIMBER is very efficient at processing XML structural queries. The implementation of "traditional" relational-style queries such as attribute value selection, value-based joins, and value-based aggregations is not highly tuned in TIMBER. This is primarily because TIMBER is a research

Discussed Section	Query Group (Queries in Group)	Geometric Mean Response Times (seconds)			
		ds0.1x		ds1x	
		TIMBER	ORDBMS	TIMBER	ORDBMS
5.2.1	Returned structure (QR1-QR4)	0.35	0.27	11.80	3.87
5.2.2	Exact match attribute value Selection (QS1-QS7)	0.06	0.04	1.30	0.28
5.2.3	Element name selection (QS8)	0.02	0.02	0.18	0.15
5.2.4	Order-based selection (QS9-QS10)	0.03	0.06	31.25	0.61
5.2.5	Element content selection (QS11-QS12)	N/A	0.12	N/A	2.75
5.2.5	String distance selection (QS13-QS14)	N/A	1.12	N/A	39.52
5.2.6	Order-sensitive selection (QS15-QS17)	0.14	0.08	0.18	0.26
5.2.7	Parent-child (P-C) selection (QS18-QS20)	0.23	0.04	2.49	0.34
5.2.7	Ancestor-descendant (A-D) selection (QS21-QS23)	0.23	2.14	2.50	17.92
5.2.7	Ancestor nesting in A-D selection (QS24-QS26)	0.24	1.16	2.70	12.65
5.2.8	P-C complex pattern selection (QS27-QS30)	0.47	0.03	5.28	0.50
5.2.8	A-D complex pattern selection (QS31-QS34)	0.44	2.49	4.91	24.74
5.2.9	Negated selection (QS35)	1.57	2.10	17.36	23.38
5.2.10	Value-based join (QJ1-QJ2)	8.90	0.05	N/A	0.42
5.2.10	Pointer-based join (QJ3-QJ4)	N/A	0.02	N/A	0.14
5.2.11	Value aggregation (QA1-QA3)	11.07	0.23	1184.69	2.31
5.2.11	Structural aggregation (QA4-QA6)	0.36	1.06	209.84	11.97

Figure 5: Benchmark Numbers for Two DBMSs

prototype and most of the development attention has been paid to the XML query processing issues that are not covered by traditional relational techniques. However, these results show that there is room for substantial performance improvement for traditional relational query operations in the current implementation of TIMBER. Structural aggregate queries are also slow in the current implementation of TIMBER. We intend to address these performance issues in the near future.

The ORDBMS can execute almost all queries well, except for the ancestor-descendant relationship queries. The ORDBMS performs very well on the parent-child queries, which are evaluated using foreign key joins.

5.2 Detailed Performance Analysis

In this section, we analyze the impact of various factors on the performance that was observed.

5.2.1 Returned Structure (QR1-QR4)

Figure 6 shows the performance of returned structure queries, QR1-QR4.

Query	Brief Query Description	Sel.(%)	Response Times (seconds)			
			ds0.1x		ds1x	
			TIMBER	ORDBMS	TIMBER	ORDBMS
QR1	Return result element	1.6	0.01	0.02	0.18	0.16
QR2	Return element and immediate children	1.6	0.34	0.31	65.00	2.59
QR3	Return entire sub-tree	1.6	10.73	1.09	387.23	26.09
QR4	Return element and selected descendants	1.6	0.32	0.97	4.27	20.57

Figure 6: Benchmark Numbers for Two DBMSs on Returned Structure Queries

As shown in Figure 6, returned structure has an impact on both TIMBER and the ORDBMS. TIMBER performs the worst when the whole subtree is returned (QR3). This is surprising since TIMBER stores elements in depth-first order, so that retrieving a sub-tree should be a fast sequential scan. It turns out that TIMBER uses SHORE [7] for low level storage and memory management, and the initial implementation of the TIMBER data manager makes one SHORE call per element retrieved. The poor performance of QR3 helped TIMBER designers identify this implementation weakness, and the TIMBER data manager is currently being modified to support sequential scan-based retrievals.

The ORDBMS takes more time in selecting and returning descendant nodes (QR3 and QR4), than returning children nodes (QR2). This is because the ORDBMS exploits the indices on the primary keys and the foreign keys when it retrieves the children nodes. On the other hand, the ORDBMS needs to call recursive SQL statements in retrieving the descendant nodes.

Simple Selection (QS1-QS10)

Figure 7 shows the performance of simple selection queries, QS1-QS10.

Query	Brief Query Description	Sel. (%)	Response Times (seconds)			
			ds0.1x		ds1x	
			TIMBER	ORDBMS	TIMBER	ORDBMS
QS1	Selection on string attribute value (low sel.)	0.8	0.01	0.02	0.09	0.08
QS2	Selection on string attribute value (high sel.)	6.3	0.06	0.06	0.71	0.63
QS3	Selection on integer attribute value (low sel.)	0.7	0.01	0.02	0.09	0.08
QS4	Selection on integer attribute value (high sel.)	6.0	0.06	0.06	0.68	0.57
QS5	Selection on range values	6.3	0.07	0.06	0.72	0.60
QS6	Selection with sorting	6.0	0.44	0.06	578.55	0.58
QS7	Multiple-attribute selection	1.6	0.32	0.02	3.65	0.17
QS8	Element name selection	1.6	0.02	0.02	0.18	0.15
QS9	Order-based selection (high fanout)	0.4	0.02	0.06	30.72	0.61
QS10	Order-based selection (low fanout)	0.4	0.04	0.06	31.79	0.62

Figure 7: Benchmark Numbers for Two DBMSs on Simple Selection Queries

5.2.2 Exact Match Attribute Value Selection (QS1-QS7)

Predicate Selectivity (QS1-QS4) Selectivity has an impact on both TIMBER and the ORDBMS. The response times of the high selectivity queries (QS2 and QS4) are more than those of the low selectivity queries (QS1 and QS3), with the response times growing linearly with the increasing selectivity for both systems. In both systems, selection on short strings is as efficient as selection on integers.

Range Selection (QS5) Both systems handle a range predicate just as well as an equality predicate. In both systems, the performance of the range predicate query (QS5) is almost the same as that of the comparable equality selection queries (QS2 and QS4).

Multiple-attribute Selection and Sorting (QS6-QS7) Currently, TIMBER does not support multiple-attribute indices. This is why it has high response times for QS6 and QS7 than for QS3. To evaluate QS6, TIMBER needs to use an unclustered index to access all nodes that satisfy the predicate aLevel = 13; then it has to retrieve the actual nodes and sort these nodes on the value of the aSixtyFour attribute. To evaluate QS7, TIMBER requires two index accesses (one for each predicate) and a set intersection between them. On the other hand, the ORDBMS performs well on QS6 and QS7, since it uses the appropriate multiple-attribute indices.

5.2.3 Element Name Selection (QS8)

Both TIMBER and ORDBMS resolve the query request for a given element name very well. This is because TIMBER uses an index on element names, and the ORDBMS simply requests all tuples from the table corresponding to the given element name.

5.2.4 Fanout (QS9-QS10)

The response times of QS9 and QS10 indicate that the small difference in fanout does not have an impact on the performance of either system. This is because TIMBER does not need to access all the children to determine the fanout; it just accesses the node in question. The ORDBMS exploits an index on the child order attribute in both cases.

5.2.5 Text Processing (QS11-QS14)

Figure 8 shows the performance of text processing queries, QS11-QS14.

Query	Brief Query Description	Sel. (%)	Response Times (seconds)			
			ds0.1x		ds1x	
			TIMBER	ORDBMS	TIMBER	ORDBMS
QS11	Element content selection (low sel.)	0.2	N/A	0.02	N/A	0.17
QS12	Element content selection (high sel.)	12.5	N/A	0.97	N/A	43.85
QS13	String distance selection (low sel.)	0.8	N/A	0.95	N/A	42.79
QS14	String distance selection (high sel.)	6.3	N/A	1.31	N/A	45.42

Figure 8: Benchmark Numbers for Two DBMSs on Text Processing Queries

Currently, TIMBER does not support content-based queries. In the ORDBMS, processing long strings (QS11-QS12) is more expensive than processing short ones (QS1-QS2) because there is no index on the long strings. The large difference between the response times of QS11 and of QS12 is due to the large difference between the scan costs of two different tables. To measure the distance between words stored in a long string (QS13-QS14), we need to use a user-defined function, which cannot make use of an index; as a result, the efficiency of the query is the same regardless of the selectivity of the string distance selection predicate.

Structural Selection (QS15-QS35)

5.2.6 Order-Sensitive Parent-Child Selection (QS15-QS17)

Figure 9 shows the performance of ordering queries, QS15-QS17.

Query	Brief Query Description	Sel. (%)	Response Times (seconds)			
			ds0.1x		ds1x	
			TIMBER	ORDBMS	TIMBER	ORDBMS
QS15	Local ordering	3.1	20.42	0.08	$>$ 1 hr	1.02
QS16	Global ordering	1 node	0.00	0.01	0.02	0.03
QS17	Reverse ordering	0.7	0.14	0.07	1.55	0.55

Figure 9: Benchmark Numbers for Two DBMSs on Ordering Queries

In TIMBER, local ordering (QS15) results in considerably worse performance than global ordering (QS16) and reverse ordering (QS17) because it requires many random accesses. On the other hand, global ordering (QS16) performs well because it requires only one random access, and reverse ordering (QS17) requires a structural join and no random access.

In the ORDBMS, local ordering (QS15) and reverse ordering (QS17) are more expensive than global ordering (QS16). This is because local ordering (QS15) needs to access a number of nodes that satisfy the given order, and reverse ordering (QS17) needs to first find the order that is the largest and then retrieve the element that has that order. On the other hand, QS16 quickly returns as soon as it finds the first tuple that satisfies the given order and predicates.

Figure 10 shows the performance of the structural selection queries, QS18-QS35. In this figure, “P-C:low-high” refers to the join between a parent with low selectivity and a child with high selectivity, whereas, “A-D:high-low” refers to the join between an ancestor with high selectivity and a descendant with low selectivity.

5.2.7 Simple Pattern Containment Selection (QS18-QS27)

The ORDBMS processes direct containment queries (QS18-QS20) better than TIMBER, but TIMBER handles indirect containment queries (QS21-QS27) better.

In TIMBER, structural joins [4] are used to evaluate both types of containment queries. Each structural join reads both inputs (ancestor/parent and descendant/child) once from indices. It keeps potential ancestors in a stack and joins them with the descendants as the descendants arrive. Therefore, the cost of the ancestor-descendant queries is not necessarily higher than the parent-child queries. From the performance of these queries, we can deduce that the higher selectivity of ancestors, the greater the delay in the query performance (QS19 and QS22). Although the selectivities of ancestors in QS20 and QS23 are lower than those of QS18 and QS21, QS20 and QS23 perform worse because of the high selectivities of descendants.

The ORDBMS is very efficient for processing parent-child queries (QS18-QS20), since these translate into foreign key joins, which the system can evaluate every efficiently using indices. On the other hand, the ORDBMS has much longer response times for the ancestor-descendant queries (QS21-QS23). The only way to

Query	Brief Query Description	Sel. (%)	Response Times (seconds)			
			ds0.1x		ds1x	
			TIMBER	ORDBMS	TIMBER	ORDBMS
QS18	P-C: medium-medium	0.7	0.14	0.02	1.51	0.25
QS19	P-C: high-low	0.7	0.29	0.05	3.21	0.44
QS20	P-C: low-high	0.7	0.28	0.05	3.18	0.34
QS21	A-D:medium-medium	3.5	0.14	2.22	1.55	20.15
QS22	A-D:high-low	0.7	0.29	0.94	3.21	5.64
QS23	A-D:low-high	1.5	0.28	4.69	3.14	50.65
QS24	Ancestor nesting in A-D:medium-medium	1.0	0.13	2.16	1.56	20.03
QS25	Ancestor nesting in A-D:high-low	1.7	0.31	0.95	3.44	8.83
QS26	Ancestor nesting in A-D:low-high	0.5	0.28	0.92	3.12	11.73
QS27	Ancestor nesting in A-D:low-high (a pair of nodes is returned)	5.0	0.29	0.97	3.19	12.35
QS28	P-C chain:high-low-low-high	0.0	0.95	0.06	10.61	0.77
QS29	P-C twig:low-high, low-low	0.0	0.31	0.02	3.57	0.48
QS30	P-C twig:high-low, high-low	0.0	0.34	0.02	3.89	0.34
QS31	A-D chain:high-low-low-high	0.4	0.95	20.22	10.56	190.19
QS32	A-D twig:low-high, low-low	0.9	0.33	2.64	3.67	33.04
QS33	A-D twig:high-low, high-low	0.4	0.34	1.24	3.84	10.86
QS34	Twig with one P-C (high-low) and one A-D (high-low)	0.2	0.34	0.58	3.90	5.49
QS35	Negated selection	93.2	1.57	2.10	17.36	23.38

Figure 10: Benchmark Numbers for Two DBMSs on Containment Selection Queries

answer these queries is by using recursive SQL statements, which are expensive to evaluate.

We also found that the performance of the ancestor-descendant queries was very sensitive to the SQL query that we wrote. To answer an ancestor-descendant query with predicates on both the ancestor and descendant nodes, the SQL query must perform the following three steps: 1) Start by selecting the ancestor nodes, which could be performed by using an index to quickly evaluate the ancestor predicate, 2) Use a recursive SQL to find all the descendants of the selected ancestor nodes, and 3) Finally, we need to check if the selected descendants match the descendant predicate specified in the query. Note that one cannot perform step 3 before step 2 since it is possible that a descendant in the result may be below another descendant that does not match the predicate on the descendant node in the query. Another alternative is to start step 1 by selecting the descendant nodes and following these steps to find the matching ancestors. In general, it is more effective to start from the descendants if the descendant predicate has a lower selectivity than the ancestor predicate. However, if the ancestor predicate has a lower selectivity, then one needs to pick the strategy that traverses fewer number of nodes. Traversing from the descendants, the number of visited nodes grows proportional to the distance between the descendants and the ancestors. However, traversing from the ancestors, the number of visited nodes can grow exponentially at the rate of the fanout of the ancestors.

The effect of the number of visited nodes in the ORDBMS is clearly seen by comparing queries QS23 and QS26. Both queries have similar selectivities on both ancestors and descendants, and are evaluated by starting from the ancestors. However, QS23 has a much higher response time. In QS23, starting from the ancestors, the number of visited nodes grow significantly since the ancestors are the nodes at level 11 – each node at this level, and its expanded non-leaf descendant nodes, has a fanout of 2. In contrast, in QS26, the ancestor set is the set of nodes that satisfy the predicate aSixtyFour = 9. Since half of these ancestors are at the leaf level, when finding the descendants, the number of visited nodes does not grow as quickly as it did in the case of query QS23.

One may wonder whether QS23 would perform better if the query was coded to start from the descendants. We found that for the ds1x data set, using this option nearly doubles the response time to 126.01 seconds. Starting from the ancestors results in better performance since the selectivity of the descendants (sel=1/4) is higher than the selectivity of the ancestors (sel=1/64), which implies that the descendent candidate list is much larger than the ancestor candidate list. Consequently, starting from the descendants results in visiting more number of nodes.

Both systems are immune to the recursive nesting of ancestor nodes below other ancestor nodes; the queries on recursively nested ancestor nodes (QS24-QS26) have the same response times as their non-recursive counterparts (QS21-QS23), except QS26 and QS23 that have different response times in the ORDBMS.

5.2.8 Complex Pattern Containment Selection (QS28-QS34)

Overall, TIMBER performs well on complex queries. It breaks the chain pattern queries (QS28 and QS31) or twig queries (QS29-QS30, QS32-QS34) into a series of binary containment joins. The performance of direct containment joins (QS28-QS30) is close to that of indirect containment joins (QS31-QS33). This is because all containment joins use efficient structural join algorithms as described in [4].

The ORDBMS takes much more time to answer ancestor-descendant chain joins (QS31-QS33) than parent-child chain joins (QS28-QS30). Again, the high response times of ancestor-descendant queries are due to the recursive SQL, which is expensive to compute. In constructing the SQL queries for the ORDBMS, we followed the techniques described in [13] for choosing the order of the joins; as expected choosing a random join order resulted in a substantial reduction in performance in some case.

As shown in the Figure 10, even though QS31 and QS33 have similar result selectivities, it is much more expensive to evaluate query QS31, in both systems. This is because QS31 has more joins than QS33. The increase in the number of joins has a greater negative effect on the ORDBMS than on TIMBER.

5.2.9 Irregular Structure (QS35)

Since some parts of an XML document may have irregular data structure, such as missing elements, queries such as QS35 are useful when looking for such irregularities. Query QS35 looks for all BaseType elements below which there is no OccasionalType element.

In TIMBER, this operation is very fast because it uses a variation of the structural joins used in evaluating containment queries. This join outputs ancestors that *do not* have a matching descendant.

In the ORDBMS, there are two ways to implement this query. A naive way is to use a set different operation which results in a very long response time (1517.4 seconds for the ds1x data set). This long response time is because the ORDBMS first needs to find a set of elements that contain the missing elements (using a recursive SQL query), and then find elements that are not in that set. The second alternative of implementing this query is to use a left outer join. That is first create a view that selects all BaseType elements have some OccasionalType descendants (this requires a recursive SQL statement). Then compute a left-outer join between the view and the relation that holds all BaseType elements, selecting only those BaseType elements that are not present in the view (this can be accomplished by checking for a null value). Compared to the response time of the first implementation (1517.4 seconds), this rewriting query results in much less response time (23.38 seconds) as reported in Figure 10.

5.2.10 Value-Based and Pointer-Based Joins (QJ1-QJ4)

Figure 11 shows the performance of value-based join queries, QJ1-QJ4.

Query	Brief Query Description	Sel. (%)	Response Times (seconds)			
			ds0.1x		ds1x	
			TIMBER	ORDBMS	TIMBER	ORDBMS
QJ1	Value-based join (low sel.)	1.6	1.73	0.03	$>$ 1 hr	0.21
QJ2	Value-based join (high sel.)	6.3	45.78	0.08	$>$ 1 hr	0.83
QJ3	Pointer-based join (low sel.)	0.02	N/A	0.01	N/A	0.05
QJ4	Pointer-based join (high sel.)	0.4	N/A	0.05	N/A	0.42

Figure 11: Benchmark Numbers for Two DBMSs on Traditional Join Queries

In TIMBER, a simple, unoptimized nested loop join algorithm is used to evaluate value-based joins. Both QJ1 and QJ2 perform poorly because of the high overhead in retrieving attribute values through random accesses. QJ3 and QJ4 require element content retrieval in joins which is not currently supported by TIMBER.

The ORDBMS performs well on this class of queries, which are evaluated using foreign-key joins which are very efficiently implemented in traditional commercial database systems.

5.2.11 Structural Aggregation vs. Value Aggregation (QA1-QA6)

Figure 12 shows the performance of aggregation queries, QA1-QA6.

Query	Brief Query Description	Sel. (%)	Response Times (seconds)			
			ds0.1x		ds1x	
			TIMBER	ORDBMS	TIMBER	ORDBMS
QA1	Value aggregation	1 node	12.12	0.01	1184.69	0.11
QA2	Value aggregation with groupby	16 nodes	10.11	0.06	N/A	0.54
QA3	Value aggregate selection	0.3	N/A	18.14	N/A	201.43
QA4	Structural aggregation	0.02	0.04	0.39	N/A	3.55
QA5	Structural aggregate selection	3.1	15.38	0.26	1288.67	3.65
QA6	Structural exploration	0.4	0.07	12.00	34.17	132.59

Figure 12: Benchmark Numbers for Two DBMSs on Aggregate Queries

In TIMBER, a native XML database, the structure of the XML data is maintained and reflected throughout the system. Therefore, a structural aggregation query, such as QA4, performs well. On the other hand, a value aggregation query, such as QA2, performs worse due to a large number of random accesses. To resolve QA2, TIMBER first retrieves the nodes sorted by level through accessing the level index, then retrieves the attributes of these nodes through random database accesses. The high response time of queries that request random accesses, such as QR3 and QA2, prompted the re-design of parts of the data manager in TIMBER to support sequential scan.

In the ORDBMS, evaluating the structural aggregation is much more expensive than evaluating the value aggregation. This is because in the relational representation the structure of XML data has to be reconstructed using expensive join operations, whereas attribute values can be quickly accessed using indices.

5.3 Performance Analysis on Scaling Databases

In this Section, we discuss the performance scaling ability of TIMBER and ORDBMS as the data set changes from **ds0.1x** to **ds1x**. Please refer to Figure 5 for the performance comparisons between these two data sets.

5.3.1 Scaling Performance on TIMBER

TIMBER is a University system still under development, and was not able to support some of the queries for **ds1x** data set, particularly those involving aggregation, join, and selection with sorting. Improving the performance of these types of queries is the focus of the TIMBER group. However, it scales well for most of the queries, particularly ancestor-descendant relationship queries and negated selection queries.

5.3.2 Scaling Performance on the ORDBMS

In most queries, the ratios of the response times when using **ds0.1x** over **ds1x** are approximately 10. Exceptions to this occur in two types of queries: a) the returned structure with descendants queries and b) the text processing queries.

As shown in Figure 6, QR3 and QR4 have response times grow about 20 times as data size increases about 10 times. This suggests that the ORDBMS needs to improve this type of queries in order to perform well on constructing an XML document from relational data. Recently, Shanmugasundaram et al. [16, 22] have addressed this problem as they proposed techniques for efficiently publishing and querying XML view of relational data. However, we were unable to leverage these techniques as they were not implemented in the ORDBMS that we used.

As shown in Figure 8, there are two types of text processing queries: element content selection (QS11-QS12), and string distance selection (QS13-QS14). Queries QS11 and QS12 are single table queries that use a `LIKE` predicate. The attribute being queried does not have an index in both data sets (the index wizard chose not to build an index on this attribute). Consequently, in both cases a full scan of the table is required.

The same behavior is seen for queries QS13 and QS14, which also use table scans. However, instead of using a `LIKE` predicate (as QS12 did), they use a user-defined function. The costs of these queries also increases faster than the table size.

6 Conclusions

We proposed a benchmark that can be used to identify individual data characteristics and operations that may affect the performance of XML query processing engines. With careful analysis of the benchmark queries, engineers can diagnose the strengths and weaknesses of their XML databases. In addition, engineers can try different query processing implementations and evaluate these alternatives with the benchmark. Thus, this benchmark is a simple and effective tool to help engineers improve system performance.

We evaluated a native XML database, TIMBER, and a commercial Object-Relational DBMS using this benchmark and found places where each had substantial room for improvement. The benchmark has already become an invaluable tool in the development of the TIMBER native XML database, helping us to identify portions of the system that need performance tuning. The benchmarking also shows that the ORDBMS is sensitive to the method used to translate an XML query to SQL. While this has been shown to be true for some XML queries in the past [13, 16], we show that this is also true for simple indirect containment queries, and queries that search for irregular structures in the XML data. We also demonstrate that using recursive SQL one can evaluate any structural query in the benchmark, however, this is much more expensive in the ORDBMS than the implementations in TIMBER, which use efficient XML structural join algorithms.

Finally, we note that the proposed benchmark meets the key criteria for a successful domain-specific benchmark that have been proposed in [15]. These key criteria are: relevant, portable, scalable, and simple. The proposed Michigan benchmark is *relevant* to testing the performance of XML engines because proposed queries are the core basic components of typical application-level operations of XML application. Michigan benchmark is *portable* because it is easy to implement the benchmark on many different systems. In fact, the data generator for this benchmark data set is freely available for download from the Michigan benchmark's web site [25]. It is *scalable* through the use of a scaling parameter. It is *simple* since it comprises only one data set and a set of simple queries, each designed to test a distinct functionality.

References

[1] A. Aboulnaga and J. Naughton and C. Zhang. Generating Synthetic Complex-structured XML Data. In *International Workshop on the Web and Databases*, Santa Barbara, California, May 2001.

[2] A. Schmidt and F. Wass and M. Kersten and D. Florescu and M. J. Carey and I. Manolescu and R. Busse. Why And How To Benchmark XML Databases. *SIGMOD Record*, 30(3), September 2001.

[3] Software AG. Tamino - The XML Power Database, 2001. `http://www.softwareag.com/tamino/`.

[4] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu. Strucutral Joins: A Primitive for Efficient XML Query Processing Pattern Matching. In *ICDE*, San Jose, CA, 2002.

[5] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene:An Extensible Templated-based Data Generator for XML. In *Fifth International Workshop on the Web and Databases*, pages 49–54, Madison, WI, 2002.

[6] T. Böhme and E. Rahm. XMach-1: A Benchmark for XML Data Management. In *Proceedings of German Database Conference BTW2001*, Oldenburg, Germany, March 2001.

[7] M. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. McAuliffe, J. F. Naughton, D. T. Schuh, and M. H. Solomon. Shoring up Persistent Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394, Minneapolis, Minnesota, 1994.

[8] IBM Corporation. DB2 XML Extender, 2001. http://www-4.ibm.com/software/data/db2/extenders/xmlext/.

[9] Microsoft Corporation. Microsoft SQL Server, 2001. http://www.microsoft.com/sql/techinfo/xml.

[10] Oracle Corporation. XML on the Oracle, 2001. http://technet.oracle.com/tech/xml/content.html.

[11] D. J. DeWitt. The Wisconsin Benchmark:Past, Present, and Future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, second edition, 1993.

[12] eXcelon Corporation. eXcelon Corporation: Products, 2001. http://www.exceloncorp.com/platform/index.shtml.

[13] M. F. Fernandez, A. Morishima, and D. Suciu. Efficient Evaluation of XML Middle-ware Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania, May 2001.

[14] R. Goldman, J. McHugh, and J. Widom. From Seminstructured Data to XML:Migrating to the Lore Data Model and Query Language. In *International Workshop on the Web and Databases*, pages 25–30, Philadelphia, Pennsylvania, June 1999.

[15] J. Gray. Introduction. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, second edition, 1993.

[16] J. shanmugasundaram and E. J. Shekita and R. Barr and M.J. Carey and B.G. Lindsay and H.Pirahesh and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. *The VLDB Journal*, 10(2-3):133–154, 2001.

[17] M. J. Carey and D. J. DeWitt and J. F. Naughton. The OO7 Benchmark. *SIGMOD Record (ACM Special Interest Group on Managment of Data)*, 22(2):12–21, 1993.

[18] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Wid om. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, September 1997.

[19] S. Bressan and G. Dobbie and Z. Lacroix and M. L. Lee and Y. G. Li and U. Nambiar and B. Wadhwa . XOO7: Applying OO7 Benchmark to XML Query Processing Tools. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, Atlanta, Georgia, November 2001.

[20] A. Sahuguet, L. Dupont, and T. L. Nguyen. Querying XML in the New Millennium. http://db.cis.upenn.edu/KWEELT/.

[21] A.R. Schmidt, F. Wass, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey, and R. Busse. The XML Benchmark Project. Technical report, CWI, Amsterdam, The Netherlands, April 2001.

[22] J. Shanmugasundaram, J. Keirnan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *Proceedings International Conference Very Large Data Bases*, pages 261–270, Roma, Italy, September 2001.

[23] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D.DeWitt, and J.Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings International Conference Very Large Data Bases*, Edinburgh, Scotland, September 1999.

[24] Poet Software. Fastobjects, 2001. http://www.fastobjects.com/FO_Corporate_Homepage_a.html.

[25] The Michigan Benchmark Team. The Michigan Benchmark: Towards XML Query Performance Diagnostics, June 2002. http://www.eecs.umich.edu/db/mbench.

[26] The TIMBER Database Team. Tree-structured native XML Database Implemented at the University of Michigan (TIMBER), 2001. http://www.eecs.umich.edu/db/timber/.