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 distribution 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 test and analyze the performance of different XML database system implementations.

1 Introduction

XML query processing has taken on considerable importance recently, and several XML databases [3, 5, 9–11, 13, 23] 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 [7, 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 [12] 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.

Another issue in this regard is the nature of the data in the test database used for the benchmark. If the data is specified to represent a particular "real application", it is likely to be quite uncharacteristic for other applications with different data distributions. 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.

Our objective is to create a single heterogeneous data set with the necessary attributes to be able to specify queries that will access precisely the type of data we wish to test. We then develop a suite of queries, each of which tests database performance with respect to a particular class of data and a particular operation.

To keep the benchmark simple, and query result sizes predictable, random number generators are used sparingly. This benchmark uses random generators for only two attribute values, and derives all other data parameters from these two generated values.

In relational systems, selectivity[1] (including join selectivity) and cardinality are the two characteristics of primary interest. In addition, there may be a few additional secondary characteristics, such as clustering and tuple/attribute size. In XML databases, besides selectivity 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. The proposed benchmark in this paper tests various types of structural characteristics in XML data. The main contributions of this benchmark are:

- The identification of the characteristics of XML data 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.

The remainder of this paper is organized as follows. Section 2 presents the related work. In Section 3, we discuss about the rationale of the benchmark data set design. In Section 4, we describe the queries of the benchmark data set. Section 5 presents the results of experimental evaluation of DBMSs using the proposed benchmark. Finally, Section 6 summarizes the contribution of this paper.

2 Related Work

Several proposals for generating synthetic XML data have been proposed [1, 6]. 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, Barbora et al. [6] proposed a template-based data generator for XML, which can generate multiple tunable data sets. On the other hand, the data generator in the proposed benchmark produces a single data set.

[1]Selectivity is really not a characteristic of the data alone, being dependent on the predicate as well. For our purposes, it is helpful to think of a specified predicate and then considering the nature of the data w.r.t. it.

Three benchmarks have been proposed for evaluating the performance of XML data management systems [7, 19, 21]. XMach-1 [7] and XMark [21] generate XML data that models data from particular Internet applications. In XMach-1 [7], 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 [16] which provides a comprehensive evaluation of OODBMS performance. The OO7 schema and instances are mapped into a Document Type Definition (DTD) and the corresponding XML data sets. The eight OO7 queries are translated into three respective languages of the query processing engines: Lore [14,17], 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 similar to the benchmark evaluated, it is hard to extrapolate the results obtained to other data sets and queries.

Gray proposed the key criteria for a successful domain-specific benchmark: relevant, portable, scalable, and simple [15]. 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, we have the data generator of this benchmark data set available on the Internet [18]. 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 with a distinct functionality test purpose.

A desiderata for a benchmark for XML databases identifies components and operations, and ten challenges that the XML benchmark should address [2]. Although their proposed benchmark is not a general purpose benchmark, it meets the challenges that test performance critical aspects of XML processing.

3 Benchmark Data Set

In this section, we first discuss 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 algorithms 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 3.1.2.

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

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 when we are computing containment relationships which include an indirect containment between ancestor and descendant and a direct containment between parent and child. It is possible to have multiple nodes at different levels satisfying the ancestor and the descendant predicates. Similarly, the fanout of the node tree 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 testing 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. In this proposal, our approach is to 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.

To study the impact of fanout, we generate the data set in the following way. There are 16 levels in the tree, and each level has a fanout of 2, except levels 5, 6, 7, and 8. Levels 5, 6, and 7 have a fanout of 13, whereas level 8 has a fanout of 1/13 (at level 8 every thirteenth node has a single child). This variation in fanout is designed to permit queries that focus isolating the fanout factor. For instance, the number of nodes is 2,704 for nodes at levels 7 and 9. Nodes at level 7 have a fanout of 13, whereas nodes at level 9 have a fanout of 2. Queries against these two levels can be used to measure the impact of fanout.

3.1.2 Data Set Granularity

To keep the benchmark simple, we choose the most aggressive choice – namely 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. Due to the limitation of presentation space, more details regarding the scaling of the benchmark data set can be found in [18].

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. Note that this attribute is set to aUnique1 + aUnique2 mod 16 instead of aUnique2 mod 16 to avoid a correlation between the predicate on this attribute and one on either aFour or aSixtyFour.
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 Figure 2.

In this section, we have described the structure of the data set. In the next section, we will examine how to generate the string content of attributes and elements in the data set.

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 the string value that has the 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) [2] 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. The words are not meaningful in any language, but simply 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

[2]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:complexType name=“BaseType” mixed=“true”>
  <xsd:sequence>
    <xsd:element name=“eNest” type=“BaseType”
      minOccurs=“0”>
      <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” maxOccurs=“1”>
      <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=“OccassionalType”>
    <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

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.

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 the evaluating the cost of individual pieces of core query functionality than in the 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 Queries

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 "Bernstein". 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%). This query is run in the following cases:

- **QR1.** Return only the elements in question, not including any sub-elements.
- **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.

Note that details about the computation of the selectivities of queries can be found in Appendix A.

The remaining queries in the benchmark simply return the selected nodes (i.e., QR1), except when explicitly specified otherwise.

4.1.2 Simple Selection Queries

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

- **Exact Match Attribute Value Selection**
  Selection based on the value of a string attribute.
  QS1. High selectivity: select nodes with aString = "Sing a song of oneB1". Selectivity is 1/16 (6.2%).

  QS2. Low selectivity: select nodes with aString = "Sing a song of oneB4". Selectivity is 1/128 (0.8%).

  Selection based on the value of an integer attribute.
  We reproduce the same selectivities as in the string attribute case.

  QS3. High selectivity: select nodes with aLevel = 13. Selectivity is 6.0%.

  QS4. Low selectivity: select nodes with aLevel = 10. Selectivity is 0.7%.

  Selection on range values.
  QS5. Select nodes with aSixtyFour between 5 and 8. Selectivity is 1/16 (6.2%).

  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 $1/64$ (1.6%).

- **Element Name Selection**
  QS8. Select nodes with the element name eOccasional. Selectivity is 1/64 (1.6%).

- **Element Content Selection**
  QS9. Select nodes that have "oneB4" as a substring of element content. Selectivity is 12.5%.

  QS10. Select OccasionalType nodes that have "oneB4" in the element content. Selectivity is 0.2%.

- **Order-based Selection**
  QS11. Select the second child of every node with aLevel = 7. Selectivity is 0.4%.

  QS12. Select the second child of every node with aLevel = 9. Selectivity is again 0.4%.

  Since the fraction of nodes in these two queries are the same, the performance difference between queries QS11 and QS12 is likely to be on account of fanout.

- **String Distance Selection**
  QS13. Select all nodes with element content that the distance between keyword "oneB2" and keyword "twenty" is not more than four. Selectivity is 6.2%.

  QS14. Select all nodes with element content that the distance between keyword "oneB5" and keyword "twenty" is not more than four. Selectivity is 0.8%.

4.1.3 Structural Selection Queries

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.

In this section, all queries return only the root of the selection pattern, unless otherwise specified.

- **Parent-child Selection**
  QS15. Medium selectivity of both parent and child. Select nodes with aLevel = 13 that have a child with aSixteen = 3. Selectivity is 0.8%.

  QS16. High selectivity of parent and low selectivity of child. Select nodes with aLevel = 15 that have a child with aSixtyFour = 3. Selectivity is 0.8%.

  QS17. Low selectivity of parent and high selectivity of child. Select nodes with aLevel = 11 that have a child with aFour = 3. Selectivity is 0.8%.

- **Order-sensitive Parent-child Selection**
  QS18. Local Ordering. Select the second element below *each* element with aFour = 1 if that second element also has aFour = 1. Selectivity is 3.1%.

  QS19. Global Ordering. Select the second element with aFour = 1 below *any* element with aSixtyFour = 1. This query returns at most one element, whereas the previous query returns one for each parent.

  QS20. Reverse Ordering. Among the children with aSixteen = 1 of the parent element with aLevel = 13, select the last child. Selectivity is 0.8%.

- **Ancestor-Descendant Selection**
  QS21. Medium selectivity of both ancestor and descendant. Select nodes with aLevel = 13 that have a descendant with aSixteen = 3. Selectivity is 3.6%.

  QS22. High selectivity of ancestor and low selectivity of descendant. Select nodes with aLevel = 15 that have a descendant with aSixtyFour = 3. Selectivity is 0.7%.

QS23. Low selectivity of ancestor and high selectivity of descendant. Select nodes with aLevel = 11 that have a descendant with aFour = 3. 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 that have a descendant with aSixteen = 3.

  QS25. High selectivity of ancestor and low selectivity of descendant. Select nodes with aFour = 3 that have a descendant with aSixtyFour = 3.

  QS26. Low selectivity of ancestor and high selectivity of descendant. Select nodes with aSixtyFour = 9 that have a descendant with aFour = 3.

  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. These two situations 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).

- **Return Structure in Structural Pattern Selection Query:**
  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 Queries:**
  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 of each of these types of queries. 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 that match the condition A, such that there is a child node that matches the condition B, such that some descendant of the child node matches the condition C. The twig query shown in the Figure 4(ii) matches all nodes that satisfy the condition A, and have a child node that satisfies the condition B, and also has a descendant node that satisfies the condition C.

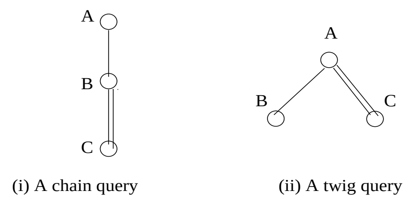

Figure 4: Sample of Chain and Twig Queries

  We use the following complex queries in our benchmark:
  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, aSixteen=3, aSixteen=5 and aLevel=16.

  QS29. One twig query with two parent-child selection (low-high, low-low). Select parent nodes with aLevel=11 (low selectivity) that have a child with aFour=3 (high selectivity), and another child with aSixtyFour=3 (low selectivity).

  QS30. One twig query with two parent-child selection (high-low, high-low). Select parent nodes with aFour=1 (high selectivity) that have a child with aLevel=11 (low selectivity) and another child with aSixtyFour=3 (low selectivity).

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

  QS34. One twig query with one parent-child selection and one ancestor-descendant selection. Select nodes with aFour=1 that have a child of nodes with aLevel=11, and a descendant with aSixtyFour = 3

- **Negated Selection Queries:**
  QS35. Find all BaseType elements below which there is no OccasionalType element.

4.2 Value-Based Join Queries

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 a tree per join-pair. Each tree has a join-node as the root, and two children, one corresponding to each element participating in the join.

QJ1. Low selectivity value join. Select nodes with aSixtyFour =2 and join with nodes with aSixtyFour = 3 based on the equality of aLevel attribute.

QJ2. High selectivity value join. Select nodes based on aSixteen =2 and join with nodes with aSixteen = 3 based on the equality of aLevel attribute.

4.3 Pointer-based Join Queries

The difference between these following queries and the join queries based on values (QJ1-QJ2) is that references which can be specified in the DTD or XML Schema and may be optimized with logical OIDs in some XML databases.

QJ3. Low selectivity pointer-based join. Select all OccasionalType nodes that point to a node with aSixty-Four =3. Selectivity is 1/4096 (0.02%).

QJ4. High selectivity pointer-based join. Select all OccasionalType nodes that point to a node with aFour =3. Selectivity is 1/256 (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 Aggregate Queries

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 for the aSixtyFour attribute of all nodes at level 15. Note that about 1/4 of all nodes are at level 15. The number of returned nodes is 1.

QA2. Value Aggregation with Groupby. Group nodes by level. 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. Structural Aggregation. Amongst the nodes at level 11, find the node(s) with the largest fanout. Selectivity is 0.02%.

QA4. Value Aggregate Selection. Select elements that have at least two occurrences of keyword "oneB1" in their content. Selectivity is 0.26%.

QA5. Structural Aggregate Selection. Select elements that have at least two children that satisfy aFour = 1. Selectivity is 3.1%.

QA6. Structural Exploration. For each node at level 7, determine the height of the sub-tree rooted at this node. The returned structure is a tree with a dummy rootthat has a child for each node at level 7. This child leaf node has one attribute that references the node at level 7, and another attribute that records the height of the sub-tree. There are only 2,704 nodes at level 7, irrespective of the database scaling. Under each of these nodes, the sub-tree goes to level 16, and so is exactly ten levels high, again irrespective of the scaling. However, determining this height may require exploring substantial parts of the database. 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. We also design update queries which can be found in [18].

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 [5], and for the ORDBMS we used the Hybrid inlining algorithm to map the data into a relational schema [22]. The queries in the benchmark were converted into SQL queries (copies of the SQL queries can be found at [18]). 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.

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 running the Windows operating system. In TIMBER, the SHORE buffer pool size was set to 100 MB. For the ORDBMS, we use the default buffer pool size since it resulted in the better execution times than when we explicitly set the buffer pool size. For ORDBMS we used their index wizard to suggest appropriate indices, which we then built. After loading the data we update all statistics, thereby giving the optimizer the most up-to-date information.

For this experiment we loaded the base data set (740K nodes, 510MB). We also wanted to load larger scaled up data sets. Surprisingly, we found that in many cases the parsers that we use for reading in the data 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. Note that because of the nature of the document tree, the percentage of nodes at the levels close to the leaves remains the same, which implies that the query selectivities stay roughly constant even in this scaled down data sets. In fact, we loaded and wrote queries against the scaled down set before using the base data set. The smaller data set size reduced the time it took for us to set up queries and load scripts for both TIMBER and the commercial ORDBMS, which we could then reuse for the base data set. We expect this strategy may be useful to other users of this benchmark too.

The results of running the benchmark on these two systems are shown in Figure 5. In this Figure, the base data set is labeled as **ds1x**. The execution times reported in this figure were produced by running each query five times, and averaging the middle three, excluding the fastest and the slowest runs. The returned results are the aUnique1 (aRef) values of the selected eNest (eOccasional) nodes.

5.2 Performance Analysis Using the ds0.1x Data Set

In this section, we analyze the impact of various factors on the performance that was observed when running the benchmark with the **ds0.1x** data set.

5.2.1 Returned structure

Returned structure has an impact on both TIMBER and the ORDBMS. TIMBER performs the worst when the whole subtree is returned (QR3). This was a surprise since TIMBER stores elements in depth-first order, so that retrieving a sub-tree should be a fast sequential scan. It turned out that TIMBER uses SHORE [8] 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 without a predicate (QR2 and QR3) than in selecting the nodes themselves (QR1) or returning descendant nodes with a predicate (QR4). The better execution times of QR2 and QR3 are due to the exploitation of indices in the ORDBMS.

5.2.2 Predicate Selectivity

Selectivity has an impact on both TIMBER and the ORDBMS. The response times of high selectivity queries (QS1 and QS3) are more than the low selectivity queries (QS2 and QS4) by 3 times in ORDBMS and 8-10 times in TIMBER. In TIMBER a query with a string predicate (QS1/QS2) performs worse than an equivalent one with an integer predicate (QS3/QS4 resp.) because it accesses a string index which is slower than an integer index. This difference is not observed in ORDBMS because the lengths of these attributes are short, and the integer index and the short string index are equally effective.

[3]while the value joins the input sizes grow linearly with the data sizes, the output sizes grow at $O(n^2) rate$

	Sel.	Response Times (seconds)			
	(%)	TIMBER		ORDBMS	
		ds0.1x	ds1x	ds0.1x	ds1x
QR1	1.6	0.02	0.11	0.03	0.20
QR2	1.6	0.33	N/A	0.53	8.66
QR3	1.6	17.10	N/A	0.98	50.93
QR4	1.6	0.76	3.74	0.25	2.33
QS1	6.2	0.23	0.43	0.06	0.72
QS2	0.8	0.03	0.06	0.02	0.13
QS3	6.1	0.10	0.50	0.06	0.62
QS4	0.8	0.01	0.06	0.02	0.13
QS5	6.1	0.09	0.48	0.06	0.64
QS6	6.1	0.56	N/A	0.06	0.64
QS7	1.6	0.69	2.60	0.03	0.20
QS8	1.6	0.07	0.13	0.02	0.02
QS9	12.5	N/A	N/A	1.14	59.44
QS10	0.2	N/A	N/A	0.02	0.92
QS11	0.4	0.04	N/A	0.06	1.16
QS12	0.4	0.04	N/A	0.06	0.64
QS13	6.3	N/A	N/A	1.35	60.01
QS14	0.8	N/A	N/A	1.06	59.00
QS15	0.8	0.29	1.82	0.03	0.46
QS16	0.8	0.72	4.98	0.05	0.69
QS17	0.8	0.50	2.53	0.06	0.70
QS18	3.1	23.74	N/A	0.09	1.66
QS19	1 node	0.00	N/A	0.02	0.09
QS20	0.8	0.36	1.90	0.08	1.74
QS21	3.6	0.31	1.96	2.47	33.95
QS22	0.7	0.72	5.01	0.74	14.80
QS23	1.5	0.48	2.33	9.57	125.94
QS24	1.0	0.30	1.90	2.46	32.92
QS25	1.7	0.74	5.19	0.75	14.97
QS26	0.5	0.48	2.28	9.56	125.09
QS27	5.0	0.50	2.69	9.70	130.83
QS28	0.0	1.91	11.88	0.07	3.46
QS29	0.0	0.58	4.03	0.06	1.09
QS30	0.0	0.81	3.58	0.05	0.66
QS31	0.5	1.93	5.61	20.35	263.40
QS32	0.9	0.51	3.91	10.20	136.69
QS33	0.5	0.81	5.62	1.18	20.19
QS34	0.2	0.81	5.77	0.73	9.57
QS35	0.9	6.03	26.58	1517.42	$>$ 14 hrs
QJ1	varies[3]	5.29	N/A	4.62	516.40
QJ2	varies[3]	99.36	N/A	73.43	8258.53
QJ3	0.02	N/A	N/A	0.01	0.17
QJ4	0.4	N/A	N/A	0.05	0.71
QA1	1 node	16.82	N/A	0.02	0.12
QA2	16 nodes	22.29	N/A	0.06	0.54
QA3	0.02	0.17	N/A	1.57	8.93
QA4	0.3	N/A	N/A	18.99	227.13
QA5	3.1	19.37	N/A	0.28	11.6
QA6	0.4	16.58	N/A	19.32	241.69

Figure 5: Benchmark Numbers for Two DBMSs

5.2.3 Range Selection

Both systems can handle range predicate as well as equality predicates efficiently. In both system, the performance of the range predicate query (QS5) is almost the same as that of the equivalent equality selection query (QS3).

5.2.4 Multiple-attribute Selection and Sorting

Currently, TIMBER does not support multiple-attribute indices. This is why it has a higher response time for QS6 and QS7 than for QS3. To evaluate QS6, TIMBER needs to access (through index) all nodes satisfying aLevel = 13 then sort them based on the value of the aSixtyFour attribute. To evaluate QS7, TIMBER performs two index accesses (one for each predicate) and performs 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.

5.2.5 Fanout

The response times of QS11 and QS12 indicate that the small difference in the 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.

5.2.6 Text Processing

Currently, TIMBER does not support content-based queries. In the ORDBMS, processing long strings (QS13 and QS14) is more expensive than processing short ones (QS1 and QS3) because there is no index on the long strings. To measure the distance between words stored in a long string, we need to build user-defined functions which are more expensive to execute than built-in functions.

5.2.7 Containment Relationships

The ORDBMS processes a direct containment queries (QS15-QS17) better than TIMBER, but TIMBER handles an indirect containment queries (QS21-QS23) better than the ORDBMS.

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 (QS16 and QS22). Although the selectivities of ancestors in QS17 and QS23 are lower than those of QS15 and QS21, QS17 and QS23 perform worse because of the high selectivities of descendants.

The ORDBMS is very efficient for processing parent-child containment queries (QS15-QS17), since the ORDBMS uses indices to evaluate these queries. On the other hand, the ORDBMS has much longer response times for the ancestor-descendant containment queries (QS21-QS23). The only way to answer these queries is by using recursive SQL statements, which is expensive to evaluate. Interestingly, the response times for these queries is ordered according to the order of the selectivities of the descendants.

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).

5.2.8 Ordering

In TIMBER, local ordering (QS18) performs considerably worse than global ordering (QS19) and reverse ordering (QS20) because it requires many random accesses. On the other hand, global ordering (QS19) performs well because it requires only one random access, and reverse ordering (QS20) requires a structural join and no random access.

In the ORDBMS, local ordering (QS18) is about as expensive as reverse ordering (QS20) while global ordering (QS19) is the least expensive. This is because local ordering (QS18) needs to access a number of nodes that satisfy the given order, and reverse ordering (QS20) needs to first find the order that is the largest and then retrieve the element that has that order. On the other hand, QS19 quickly returns as soon as it finds the first tuple that satisfies the given order and predicates.

5.2.9 Complex Pattern Selectivity

Overall, TIMBER performs well in 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 structural join [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. As before, the response times of these queries are very sensitive to the selectivities of the descendants. The response time increases as the selectivity increases.

TIMBER performs better on queries with low selectivity of ancestors (QS32) than those with high selectivity of ancestors (QS33). On the other hand, ORDBMS executes better on queries with low selectivities of descendants (QS33) than those with high selectivity of descendants (QS32).

5.2.10 Irregular Structure

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, this query takes a very long time to finish because the ORDBMS first needs to find a set of elements that contain the missing elements (a recursive SQL query is used for this part), and then find elements that are not in that set (using a set difference operation). This is inefficient compared to the structural join algorithm that TIMBER uses.

5.2.11 Join Selectivity

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 which is not currently supported by TIMBER.

In the ORDBMS, QJ2 is the most expensive join query since this query has the highest result selectivity.

5.2.12 Structural Aggregation vs. Value Aggregation

An example of structural aggregation query is QA3 which finds the nodes at level 11 with the largest fanouts. An example of value aggregation query is QA2 which computes the average value of the aSixtyFour attribute of nodes at each level. The numbers of nodes returned in QA2 and QA3 are about the same.

In TIMBER, a native XML database, the structure of the XML is maintained and reflected throughout the system. Therefore, a structural aggregation query, such as QA3, 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, the ratio of the response times of QA3 over QA2 is about ten which reflects that 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 about the performance scaling ability of TIMBER and ORDBMS as the data set changes from **ds0.1x** to **ds1x**.

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 and join. However, for the remaining queries, it scales very well, giving ratios of response times when using **ds1x** over when using **ds0.1x** data sets of 2-7. Since the **ds1x** data set is ten times larger than the **ds0.1x** data set, this represents sublinear scaling. This is probably due to significant constant overhead that is independent of data set size, since asymptotic analysis suggests that most TIMBER algorithms will scale linearly. Locating and removing this overhead should be an engineering objective for TIMBER.

5.3.2 Scaling Performance on the ORDBMS

As shown in Figure 5, in most queries, the ratios of the response times when using **ds0.1x** over when using **ds1x** are approximately 10-20. Exceptions to this occur in five types of queries: a) the complex returned structure query (QR3), b) the element content selection queries (QS9 and QS10), c) the string distance selection queries (QS13 and QS14), d) the value join queries (QJ1-QJ2), and e) the missing elements query (QS35).

For QR3, when the ORDBMS2 uses a different query plans for the two data sets. The query plan in **ds0.1x** uses a hash join algorithm for one of its query, however, with the **ds1x** data set the optimizer chooses to use nested loops join (we confirmed that changing the buffer size does not cause the optimizer to switch the query plans). Consequently, for QR3, the execution time increase sharply in the **ds1x** case. Queries QS9 and QS10 are single table queries that use a like predicate. The attribute being queried does not have an index in either of the the queries or the data sets. Nevertheless, the cost of these queries increases rapidly in the **ds1x** data set. The same behavior is seen for queries QS13 and QS14, which too use table scans, but instead of using a like predicate (as QS9 and QS10 did), use a user-defined function. For the value join queries, the result selectivity does not grow linearly, even though the input sizes do, which leads to the high response times with the **ds1x** data set. Finally, the missing elements query QS35, uses a recursive SQL query and a set difference query. The recursive part of this query finds all eNest ancestors of the eOccasional nodes. The results of this subquery is then used in a set-difference query to find all eNest elements that do not have any nested eOccassional descendant. The evaluation time to compute the recursive part of this query rises very sharply in the **ds1x** data set.

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, and, thus, be able to improve the performance.

We evaluated a native XML database, TIMBER, and a commercial Object-Relational database 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 bad implementation on our part (for instance, TIMBER's poor performance for a linear scan in QR3).

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 *Proceedings IEEE International on Data Engineering*, San Jose, CA, 2002.

[5] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, K. Thompson, and Y. Wu. Tree-structured native XML Database Implemented at the University of Michigan (TIMBER), 2001. `http://www.eecs.umich.edu/db/timber/`.

[6] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene:An Extensible Templated-based Data Generator for XML. `http://cs.toronto.edu/tox/toxgene/`.

[7] T. Böohme and E. Rahm. XMach-1: A Benchmark for XML Data Management. In *Proceedings of German Database Conference BTW2001*, Oldenburg, Germany, March 2001. `http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html`.

[8] 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.

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

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

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

[12] 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.

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

[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] 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.

[17] 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.

[18] K. Runapongsa, J. M. Patel, and H.V. Jagadish. The Michigan Benchmark: Towards XML Query Performance Diagnostics, February 2002. `http://www.eecs.umich.edu/db/mbench`.

[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. `http://monetdb.cwi.nl/xml/index.html`.

[22] 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*, page 30, Edinburgh, Scotland, September 1999.

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

A Query Selectivity Explanation

Each of the benchmark queries was carefully chosen to have a desired selectivity. In this appendix, we describe the computation of these selectivities, analytically.

For this purpose, we will frequently need to determine the probability of "PickWord", based on the uniform distribution of buckets and words in each bucket, as described in Section 3.3. For example, if "PickWord" is "oneB1", this indicates that this "PickWord" is the first word in bucket 1. Since there are 16 buckets, and there is only one word in the first bucket the probability of "oneB1" being picked is 1/16 $(1/16 \times 1)$. Since there are eight words in the fourth bucket (2^{4-1}), the probability of "oneB4" being picked is 1/128 $(1/16 \times 1/8)$.

QR1-QR4. Select all elements with aSixtyFour =1. These queries have a selectivity of 1/64 since they are selected based on aSixtyFour attribute which has a probability of 1/64.

QS1. Select nodes with aString = "Sing a song of oneB1". Selectivity is 1/16 since the probability of "oneB1" is 1/16.

QS2. Select nodes with aString = "Sing a song of oneB4". Selectivity is 1/16 since the probability of "oneB4" is 1/128.

QS3. Select nodes with aLevel = 13. Selectivity is 6.0% since the number of nodes at level 13 is 6.0% of the number of total nodes in the document.

QS4. Select nodes with aLevel = 10. Selectivity is 0.7% since the number of nodes at level 10 is 0.7% of the number of total nodes in the document.

QS5. Select nodes with aSixtyFour between 5 and 8. Selectivity is $4 \times 1/64 = 1/16$.

QS6. Select nodes with aLevel = 13 and have the returned nodes sorted by aSixtyFour attribute. Selectivity is 6.1%.

QS7. Select nodes with attributes aSixteen = 1 and aFour = 1. Selectivity is $1/16 \times 1/4 = 1/64$ (1.6)%.

QS8. Select nodes based on the element name, eOccasional. Selectivity is $1/64$ since eOccasional appears nested under the element with aSixtyFour = 0 and the probability of aSixtyFour = 0 is 1/16.

QS9. Select nodes that have "oneB4" as substring in element content. The probability of "oneB4" being picked 1/128. Although this string can also arise in bucket 16, the probability there is much smaller 2^{-19}, and hence can be ignored. There are 16 "PickWord"s in the content of each element, so 16 opportunities for this predicate to be satisfied at each element, giving an overall selectivity of 16/128 (12.5%).

QS10. Select OccasionalType nodes that have "oneB4" in the element content. Since there is approximately one OccasionalType node in every 64 BaseType nodes, the overall selectivity is $(16/128) \times (1/64) = 1/512$ (0.2%).

QS11. Select the second child of every node with aLevel = 7. At level 7, each node has thirteen children. These children are at level 8 which has the number of nodes = 4.8% of all nodes. Thus, the selectivity of this query is 4.8% × 1/13 = 0.4%.

QS12. Select the second child of every node with aLevel = 9. At level 8, each node has two children. These children are at level 10 which has the number of nodes = 0.7% of all nodes. Thus, the selectivity of this query is 0.7% × 1/2 = 0.4%.

QS13. Select all nodes with element content that the distance between keyword "oneB2" and keyword "twenty" is not more than four. There are two occurrences of "PickWord" within four words of "twenty" and 14 occurrences that are further away. The probability of any one occurrence of "oneB2" being selected is 1/32. Thus, the overall selectivity is $(1/32) \times 2 = 1/16$ (6.2%).

QS14. Select all nodes with element content that the distance between keyword "oneB5" and keword "twenty" is not more than four. The probability of any one occurrence of "oneB5" being selected is 1/256. Thus, the overall selectivity is $(1/256) \times 2 = 1/128$ (0.8%).

QS15. Select nodes with alevel = 13 that have a child with aSixteen = 3. Both the first predicate and the second predicate have a selectivity of 1/16. Since each node at level 13 has two children, there are two opportunities to satisfy the child predicate. Therefore the overall selectivity of this query is $(1/16) \times (1/16) \times 2 = 1/128$ (0.8%).

QS16. Select nodes with aLevel = 15 that have a child with aSixtyFour = 3. The first predicate and the second predicate have a selectivity of 1/4 and 1/64 respectively. Following the same argument as above, the selectivity of the query as a whole is still 1/128 (0.8%).

QS17. Select nodes with aLevel = 11 that have a child with aFour = 3. The first predicate and the second predicate have a selectivity of 1/64 and 1/4 respectively. Following the same argument as above, the selectivity of the query as a whole is still 1/128 (0.8%).

QS18. Select the second element below each element with aFour = 1 if that second element also has aFour = 1. Let n_l is the number of nodes at level l and f_{l-1} is the number of fanout at level $l-1$. Then, the number of the second element nodes is $\sum_{l=2}^{l=16}(n_l) \times (1/f_{l-1}) \approx 1/2$. Since the selectivity of the element with aFour =1 is 1/4, then probability that the second element that has aFour = 1 and that its parent has aFour = 1 is 1/16. Thus, the overall selectivity of this query is $(1/2) * (1/16) = 1/32$ (3.1%).

QS19. Select the second element with aFour = 1 below any element with aSixtyFour = 1. This query returns at most one element.

QS20. Among the children with aSixteen = 1 of the parent element with aLevel = 13, select the last one. Approximately 1/16 of the nodes are at level 13, and each has two children. Almost 1/8 of these will have at least one child that satisfies the former predicate (from among whom the last one must be returned in this query). Thus, the overall query selectivity is 1/128 (0.8%).

QS21. Select nodes with aLevel = 13 that have a descendant with aSixteen = 3. The first predicate has selectivity of 0.06. Since each node at level 13 has 14 descendants, the probability that none of these 14 nodes satisfy the second predicate is $1 - (1/16)^{14}$. Thus, the probability that a given selected ancestor node has any descendant that satisfies the second predicate is $1 - (1 - (1/16)^{14})$. Therefore, the overall selectivity is $0.06 \times (1 - (1 - (1/16))^{14})$ = 3.6%.

QS22. Select nodes with aLevel = 15 that have a descendant with aSixtyFour = 3. The first predicate has selectivity of 0.24. Since a node at level 15 only has no descendant other than its own two children, the probability that none of these two nodes satisfy the second predicate is $1 - (1/64)^2$. The overall selectivity is $0.24 \times (1 - (1 - (1/64))^2)$ = 0.7%.

QS23. Select nodes with aLevel = 11 that have a descendant with aFour = 3. The first predicate has selectivity of 1.5. Since each level 11 node has 62 descendants, the probability that none of these 62 nodes satisfy the second predicate is $1 - (1/4)^{62}$. The selectivity of the query as a whole is $1.5 \times (1 - (1 - (1/4))^{62})$ = 1.5%.

QS28. This 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, aSixteen=3, aSixteen=5 and aLevel=16. The probability of aFour = 3 is 1/4, and of aSixteen = 3(5) is 1/16, and the probability of aLevel = 16 is 0.47. Thus, the selectivity of this query is $1/4 \times 1/16 \times 1/16 \times 0.47$ = 0.0%.

QS29. Select parent nodes with aLevel=11 that have a child with aFour=3, and another child with aSixtyFour=3. The probability of aLevel=11 is 0.015, that of aFour=3 is 1/4, and that of aSixtyFour=3 is 1/64. Thus, the selectivity of this query is $0.015 \times 1/4 \times 1/64$ = 0.0%.

QS30. Select parent nodes with aFour=1 that have a child with aLevel=11 and another child with aSixtyFour=3. The probability of aFour=1 is 0.25, that of aLevel=11 is 0.015, and that of aSixtyFour=3 is 1/64. Thus, the selectivity of this query is $0.25 \times 0.015 \times 1/64$ = 0.0%.

QS32. Select nodes with aLevel = 11 that have a descendant with aFour =3 and another descendant with aSixtyFour = 3. The first predicate has selectivity of 0.015. Since a node at level 11 has 62 descendants. The probability that none of these descendants satisfy aFour=3 and aSixtyFour = 3 are $(1 - (1 - 1/4))^{62}$ and $((1 - (1 - 1/64))^{62})$, respectively. Thus, the overall selectivity is $0.015 \times (1 - (1 - (1/4))^{62} \times (1 - (1 - 1/64))^{62}$ = 0.9%.

QJ1. Select nodes with aSixtyFour = 2 and join with nodes with aSixtyFour = 3 based on the equality of aLevel attribute. The probability that a node satisfies either aSixtyFour predicate value is 1/64. Let N_l be the number of nodes at level l, then the estimated number of nodes that satisfy the predicates of this query is $\sum_{l=1}^{16}(N_l/64)^2$.

QJ2. Select nodes with aSixteen = 2 and join with nodes with aSixteen = 3 based on the equality of aLevel attribute. The probability that a node satisfies either aSixteen predicate value is 1/16. Let N_l be the number of nodes at level l, then the estimated number of nodes that satisfy the predicates of this query is $\sum_{l=1}^{16}(N_l/16)^2 4$.

QJ3. Select all OccasionalType nodes that point to a node with aSixtyFour =3. This query returns 1/64 of all the OccasionalType nodes, and the probability of OccasionalType nodes is 1/64. Thus, the selectivity of this query is $1/64 \times 1/64 = 1/4096$ (0.02%).

QJ4. Select all OccasionalType nodes that point to a node with aFour =3. This query returns 1/4 of all the eOccasional nodes, and the probability of OccasionalType nodes is 1/64. Thus, the selectivity of this query is $1/4 \times 1/64 = 1/256$ (0.4%).

QA1. Compute the average value for the aSixtyFour attribute for all nodes at level 15. This query returns only one node which contains the average value.

QA2. Compute the average value for the aSixtyFour attribute for all nodes at each level. This query returns 16 nodes which contains the average values for 16 levels.

QA3. Amongst the nodes at level 11, find the node(s) with the largest fanout. 1/64 of the nodes are at level 11. Most nodes at this level have exactly two children. But 1/64 of these nodes also have a third child, of type eOccasional. These are the nodes that must be returned. Thus, selectivity is $1/64 \times 1/64 = 1/4096$ (0.02%).

QA4. Select elements that have at least two occurrences of keyword "oneB1" in their content. There are 16 "PickWord"s in the element content. The probability that "PickWord" is replaced with "oneB1" is 1/16, and the probability that "PickWord" is not replaced with "oneB1" is 15/16. Let $P_n(\text{“oneB1''})$ be the probability that there are n occurrences of "oneB1." Then, $P_n(\text{“oneB1''})$ = $\binom{16}{n} \times (1/16)^n \times (15/16)^{16-n}$. The probability that there are at least two occurrences of "oneB1" is $1 - P_0(\text{“oneB1''}) - P_1(\text{“oneB1''})$ = 1 - 0.36 - 0.38 = 0.26. Thus, the selectivity of this query is 0.26%.

QA5. Select elements that have at least two children that satisfy aFour = 1. About 50% of the database nodes are at level 16 and have no children. Except about 2% of the remainder, all have exactly two children, and both must satisfy the predicate for the node to qualify. The selectivity of the predicate is 1/4. So the overall selectivity of this query is $(1/2) \times (1/4) \times (1/4) = 1/32$ (3.1%)

QA6. For each node at level 7, determine the height of the sub-tree rooted at this node. Nodes at level 7 are 0.4% of all nodes, thus the selectivity of this query is 0.4%.