The Michigan Benchmark: A Micro-Benchmark for XML Query Performance Diagnostics

Jignesh M. Patel and H. V. Jagadish
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109, USA
{jignesh, jag}@eecs.umich.edu

1 Introduction

With the increasing popularity of the eXtensible Markup Language (XML) as a representation format for a wide variety of data, and it is clear that large repositories of XML data sets will soon emerge. The effective management of XML in a database thus becomes a pressing issue. Several methods for managing XML databases have emerged, ranging from retrofitting commercial RDBMSs to building native XML database systems. There has naturally been an interest in benchmarking the performance of these systems, and a number of benchmarks have been proposed [4, 10, 12]. 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 [5] provided the database community with an invaluable engineering tool to assess the performance of individual operators and access methods. Inspired by the simplicity and the effectiveness of the Wisconsin benchmark for measuring and understanding the performance of relational DBMSs, we develop a comparable benchmarking tool for XML data management systems. The benchmark that we propose is called the Michigan benchmark.

A challenging issue in designing any benchmark is the choice of the data set that is used by 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.

For a micro-benchmark, 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 the queries must also guide the benchmark user to the precise component of the system that is performing poorly. We attempt to achieve this balance by using a data set that has a simple schema. 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. In addition, 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. In addition, 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 remainder of this chapter is organized as follows. Section 2 presents the related work. In Section 3, we discuss the rationale of the benchmark data set design. In Section 4, we describe the queries of the benchmark data set. Section 5 presents our recommendation on how to analyze and present the results of the benchmark. Finally, Section 6 summarizes the contribution of this benchmark.

2 Related Work

Several proposals for generating synthetic XML data have been proposed [1, 3]. Aboulnaga et al. [1] proposed a data generator that accepts as many as twenty 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. [3] 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 the Michigan 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 [4, 10, 12]. XMach-1 [4] and XMark [12] generate XML data that models data from particular Internet applications. In XMach-1 [4], the data is based on a web application that consists of text documents, schema-less data, and structured data. In XMark [12], the data is based on an Internet auction application that consists of relatively structured and data-oriented parts. XOO7 [10] is an XML version of the OO7 Benchmark [8] 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 [6, 9], Kweelt [11], 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.

A desiderata document [2] for a benchmark for XML databases identifies components and operations, and ten challenges that the XML benchmark should address. 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 the 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 the XML paradigm, there are several complicating characteristics to consider as discussed in Section 3.1.1 and 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 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 measure the effect of 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.

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

3.1.2 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.

3.1.3 Scaling

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.

The default data set, which is described in Section 3.1.1, is called **DSx1**. This data set has about 728K nodes, arranged in a tree of a depth of 16 and a fanout of 2 for all levels except levels 5, 6, 7 and 8, which have fanouts of 13, 13, 13, 1/13 respectively. From this data set we generate two additional "scaled-up" data sets, called **DSx10** and **DSx100** such that the numbers of nodes in these data sets are approximated 10 and 100 times the number of nodes in the base data set, respectively. We achieve this scaling factor by varying the fanout of the nodes at levels 5-8. For the data set **DSx10** levels 5–7 have a fanout of 39, whereas level 8 has a fanout of 1/39. For the data set **DSx100** levels 5–7 have a fanout of 111, whereas level 8 has a fanout of 1/111. The total number of nodes in the data sets **DSx10** and **DSx100** is 7,180K and 72,350K respectively (which translates into a scale factor of 9.9x and 99.4x respectively).

In the design of the benchmark data set, 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 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.

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 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]

[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: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

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 get exactly 2^{15} words in the sixteenth bucket, we add the dummy word “oneB0ing”).

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

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 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

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 at [13].

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 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. 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%.

  Selection based on the value of an integer attribute.
  We reproduce the same selectivities as in the string attribute case.
  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. Select the second child of every node with aLevel = 7. Selectivity is 0.4%.

  QS10. 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 queries QS9 and QS10 is likely to be on account of fanout.

- **Element Content Selection**
  QS11. Select OccasionalType nodes that have "oneB4" in the element content. Selectivity is 0.2%.

  QS12. 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.

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 approximately 0.7%.

  QS16. High selectivity of parent and low selectivity of child. Select nodes with aLevel = 15 that have a child with aSixtyFour = 3. Selectivity is approximately 0.7%.

  QS17. Low selectivity of parent and high selectivity of child. Select nodes with aLevel = 11 that have a child with aFour = 3. Selectivity is approximately 0.7%.

- **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 0.7%.

- **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 3.5%.

  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 = 5.

  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).

  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 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.

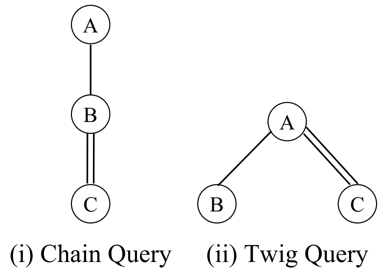

Figure 4: Samples of Chain and Twig Queries

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, 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).

- **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 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**
  QS35. Find all BaseType elements below which there is no Occasional-Type 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 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. Select nodes with aSixtyFour =2 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 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

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. Select all OccasionalType nodes that point to a node with aSixtyFour =3. Selectivity is 0.02%.

QJ4. High selectivity. Select all OccasionalType nodes that point to a node with aFour = 3. 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 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. Value aggregate selection. Select elements that have at least two occurrences of keyword "oneB1" in their content. Selectivity is 0.3%.

QA4. Structural aggregation. Amongst the nodes at level 11, find the node(s) with the largest fanout. 1/64 of the nodes are at level 11. Selectivity is 0.02%.

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 (have aLevel= 7, 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 Updates

- **QU1. Point Insert.** Insert a new node BaseType node below the node with aUnique1 = 10102. The new node has attributes identical to its parent, except for aUnique1, which is set to some new large, unique value.

- **QU2. Point Delete.** Delete the node with aUnique1 = 10102 and transfer all its children to its parent.

- **QU3. Bulk Insert.** Insert a new BaseType node below each node with aSixtyFour = 1. Each new node has attributes identical to its parent, except for aUnique1, which is set to some new large, unique value.

- **QU4. Bulk Delete.** Delete all leaf nodes with aSixteen = 3.

- **QU5. Bulk Load.** Load the original data set from a (set of) document(s).

- **QU6. Bulk Reconstruction.** Return a set of documents, one for each sub-tree rooted at level 11 (have aLevel=11) and with a child of type eOccasional.

- **QU7. Restructuring.** For a node u of type eOccasional, let v be the parent of u, and w be the parent of v in the database. For each such node u, make u a direct child of w in the same position as v, and place v (along with the sub-tree rooted at v) under u.

5 Using the Benchmark

Since the goal of this benchmark is to test individual XML query operations, we do not propose a single benchmark number that can be computed from the individual query execution times. While having a single benchmark number can be very effective in summarizing the performance of an application benchmark, for a non-application specific benchmark, such as this benchmark, it may be meaningless.

Similarly, it may be useful to run the benchmark queries in both *hot* and *cold* modes, corresponding to running the queries using a buffer pool that is warmed up by a previous invocation of the same query, and running the query with no previously cached data in the buffer pool respectively.

Group	**Group Description**	**Queries**
A	Returned Structure	QR1-QR4
B	Exact Match Attribute Value Selection	QS1-QS7
C	Element Name Selection	QS8
D	Order-Based Selection	QS9-QS10
E	Element Content Selection	QS11-QS12
F	String Distance Selection	QS13-QS14
G	Parent-Child Selection	QS15-QS17
H	Order-Sensitive Parent-Child Selection	QS18-QS20
I	Ancestor-Descendant Selection	QS21-QS23
J	Ancestor Nesting in Ancestor-Descendant Selection	QS24-QS26
K	Parent-Child Complex Pattern Selection	QS27-QS30
L	Ancestor-Descendant Complex Pattern Selection	QS31-QS34
M	Negated Selection	QS35
N	Value-Based Join	QJ1-QJ2
O	Pointer-Based Join	QJ3-QJ4
P	Value-Based Aggregation	QA1-QA3
Q	Structural Aggregation	QA4-QA6
R	Point Updates	QU1-QU2
S	Bulk Updates	QU3-QU7

Figure 5: Benchmark Groups

In our own use of the benchmark, we have found it useful to produce two 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. For the summary table, we use the groups that are shown in Figure 5. For each group, we compute the geometric mean of the execution times of the queries in that group. When comparing different systems, or when evaluating the scalability of a system using the benchmark, the summary table quickly identifies the key strengths and weaknesses of the system(s) being evaluated. The detailed table then provides more precise information on the performance of the individual query operations. We expect that this approach of using two tables to summarize the benchmark results, will also be useful to other users of this benchmark.

6 Conclusions and Future Work

The Michigan benchmark that is described in this chapter, is a micro-benchmark that can be used to tune the performance of XML query processing systems. In formulating this benchmark we paid careful attention to the techniques that we use in generating the data and the query specification, so as to make it very easy for a benchmark user to identify any performance problems. The data generation process uses random numbers sparingly and still captures key characteristics of XML data sets, such as varying fanout and depth of the data tree. The queries are carefully chosen to focus on individual query operations and to demonstrate any performance problems related to the implementation of the algorithms used to evaluate the query operation. With careful analysis of the benchmark results, engineers can diagnose the strengths and weaknesses of their XML databases, and quantitatively examine the impact of different implementation techniques, such as data storage structures, indexing methods, and query evaluation algorithms. The benchmark can also be used to examine the effect of scaling up the database size on the performance of the individual queries. In addition, the benchmark can also be used to compare the performance of various primitive query operations across different systems. Thus, this benchmark is a simple and effective tool to help engineers to be able to improve the performance of XML query processing engines.

In designing the benchmark, we paid careful attention to the key criteria for a successful domain-specific benchmark that have been proposed in [7]. 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 [13]. 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.

We are continuing to use the benchmark to evaluate a number of native XML data management systems, and traditional (object) relational database systems. We plan on publishing the most up-to-date results using the benchmark, at the web site for this benchmark [13].

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] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] The Michigan Benchmark Team. The Michigan Benchmark Homepage. `http://www.eecs.umich.edu/db/mbench`.