The Michigan Benchmark: Towards XML Query Performance Diagnostics

October 20, 2001

This benchmark is designed to help engineers design better XML query processing engines. The data set and queries of this benchmark are described in the following sections.

1 Benchmark Data Set

In this section, we first discuss the data characteristics of XML data sets that can have a significant impact on the performance of query operations. Then, we present the benchmark data sets and explain our rationale for the data characteristic parameters that we use in the benchmark data.

1.1 A Discussion of the Data Characteristics

In a relational situation, once a relational scheme is fixed, the tuple size is known, and 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 case, there are several complicating characteristics to consider, and we address these in turn below.

1.1.1 Structural Selection

Selection in XML is often based on patterns. Queries should be constructed to consider multi-node patterns of various sorts, and with various selectivities. An issue that arises is that of "conditional selectivity." Consider a simple two node selection pattern. Given that one of the nodes has been identified, conditioned upon this fact, the selectivity of the second node can be very different than its selectivity in the database as a whole. (One could, of course, have similar dependencies between different attributes in a relation, thereby affecting the selectivity of a multi-attribute predicate. Matters are more complicated in XML because these different attributes may not be in the same tuple, but rather in different, but structurally related, elements.) These dependencies can be modeled by having the "cyclic numbering" schemes with different moduli, as in the Wisconsin benchmark.

Two other data structural parameters are important for tree-structured data, such as XML. These are fanout and depth. Let us look at the issue of depth first. The depth of the data tree can have a significant performance impact when we are computing ancestor-descendant (or containment) relationships. It is possible that the containment is very distant (not an immediate child), and it is also possible to have multiple nodes at different levels satisfying the ancestor and the descendant predicates. Similarly the fanout of the data tree can have an impact on the way in which the DBMS stores the data in the DBMS, and answers queries that are based on selecting children in a specific order (for example select the last child).

One potential way of testing these parameters (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 we have to deal with a large number of data sets which makes the benchmark harder to run and to understand. Rather, the approach that we take in this proposal is to create a base benchmark data set of depth 16. Then, using a "level" attribute, 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. Each level in the tree (recall there are 16 levels) has a fanout of 2, except for levels 5, 6, 7, and 8. The levels 5, 6, and 7 have a fanout of 4, where as level 8 has a fanout of 1/4 (at level 8 every fourth node has a single child). This variation in fanout is designed to permit queries that focus on this one factor in isolation. For instance, the number of nodes at levels 7 and 9 is 256. Nodes at level 7 have a fanout of 4, whereas the nodes at level 9 have a fanout of 2. Queries against these two levels can be used to measure the impact of fanout.

1.1.2 Document size

An XML document is tree-structured. If this document is stored in a file system, XML documents themselves may exist in a tree-structured directory. The question is what constitutes an XML

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	4	16	0.0
6	4	64	0.1
7	4	256	0.4
8	0.25	1,024	1.5
9	2	256	0.4
10	2	512	0.8
11	2	1,024	1.5
12	2	2,048	3.1
13	2	4,096	6.1
14	2	8,192	12.3
15	2	16,384	24.6
16	–	32,768	49.2

Figure 1: Distribution of the nodes in the base data set

database? At one extreme, it could be a single huge tree. At the other, it could comprise many small documents. Once again, 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 level 13 node 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. We have chosen not to include this experiment in the benchmark due to our desire for simplicity, and our belief that the document granularity issue is not a critical one to study.

1.1.3 Scaling

A good benchmark should scale so that appropriately scaled versions of the same benchmark can be run on platforms with widely varying capabilities. In the relational model, scaling a benchmark data set is easy – we simply increase the number of tuples. In the XML world, there are many choices one could make, and these could have an effect on the structure of the data set. We would like to isolate the effect of the number of nodes from effects due to other structural changes, such as depth, fanout, etc. To minimize such cross-impacts, we propose to scale the Michigan benchmark in discrete steps, as discussed below.

The default data set, called **DSx1**, has 67K nodes, arranged in a tree of depth 16 and fanout 2 for all the levels except levels 5, 6, 7 and 8, which have fanouts of 4, 4, 4, 1/4 respectively. From this data set we generate two additional "scaled-up" data sets, called **DSx10** and **DSx100** that contain approximately 10 and 100 times respective to the number of nodes in the base data set. 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 13, whereas level 8 has a fanout of 1/13. For the data set **DSx100** levels 5–7 have a fanout of 38, whereas level 8 has a fanout of 1/38. The total number of nodes in the data sets **DSx10** and **DSx100** is 727K and 6,800K respectively [1].

In the design of the benchmark data set, we deliberately chose to 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

[1]this translates into a scale factor of 10.9x and 101.9x.

predicate aLevel $= 13$.

1.2 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 value 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 value generated randomly.

3. aLevel: An integer value to store the level of the node.

4. aFour: set to aUnique2 mod 4

5. aSixteen: set to aUnique2 mod 16

6. aSixtyFour: set to aUnique2 mod 64

7. aString: A string value approximately 32 bytes in length.

The content of each BaseType element is a large string that is approximately 512 bytes in length. The generation of the element content and the string attribute aString is described in Section 1.3.

In addition to the attributes listed above, each BaseType element has two sets of subelements. The first subelement is of type BaseType. The number of repetitions of this element is used to control the fanout, as described in Fig. 1. The second subelement is of type OccasionalType, and can occur either 0 or 1 times. The presence or absence of the OccasionalType element is determined by the value of the attribute aSixtyFour. A BaseType element has a nested (leaf) element of type OccasionalType if the aSixtyFour attribute has the value 0. This leaf element has content that is identical to the content of its parent. The attribute aRef is a reference to the node with id value equal to the parent's aUnique1-11. In case such an ID does not exist (for the top few nodes in the tree), the attribute points to the root of the tree.

The XML Schema [2] specification of the benchmark data is shown in Figure 2:

```
<?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="@aUnique1Ref"/>
      </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="aUnique1Ref" 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:element name="root" type="BaseType"/>
</xsd:schema>
```

Figure 2: Benchmark Specification in XML Schema

1.3 Generating the String Attributes and Element Content

The content of the elements in the benchmark is a long string. Since this long 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. Yet, selecting pieces of text from real sources introduces all manners of difficulty – how to maintain roughly constant size for each string, how to avoid idiosyncrasies associated with the specific source, how to generate more strings as required for a scaled benchmark, and so on. Moreover, we would like to have benchmark results applicable to a wide variety of languages and domain vocabularies.

To address the above desiderata, we generate these long strings synthetically, in a carefully stylized manner. We begin by creating a pool of $2^{16} - 1$ (a little over sixty thousand) words[2] synthetic words as follows. The words are divided into 16 buckets, with exponentially growing bucket occupancy. The first bucket has only one word, the second has two words, the third has 4, and so on. Bucket i has 2^{i-1} words. The words are not words from the English (or any other) language. Rather, they are synthetically created, with each word containing 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 at roughly 30,000, words in the last bucket are derived from words in the other buckets by adding the suffix "ing" (to get exactly 2^{15} words in the sixteenth bucket, we add the dummy word "oneB0ing").

The value for the long string is generated from the following template, where "PickWord" is actually a placeholder for words 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 chosen from within the 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 string attribute is simply the first line of the large string that is stored as the element content.

[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.

```
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

2 Benchmark queries

In creating the data set above, we have taken care to make it possible to tease apart data with different characteristics, and to issue queries with well-controlled but vastly differing data access patterns. Our belief is that we should do the same with respect to queries. Specifically, we are less interested in the composite performance of queries that are of application-level value. Rather, our focus is on evaluating the cost of individual pieces of core query functionality. Knowing these costs, 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 as 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 being stressed by each benchmark query. A complex query that involves many of these simple operations can be expected to take time that varies monotonically with the time required for these simple components.

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

2.1 Selection Queries

Whereas relational selection identifies the tuples that satisfy a given predicate over its attributes, XML selection is considerably more complex, as we have seen above. One is typically interested in identifying the portion of the tree that satisfies a specified predicate. Furthermore, the predicate itself could involve conditions on the attributes (or content) of several structurally related elements. Thus simply understanding the performance of (complex) selection queries can be extremely valuable in the case of an XML database.

2.1.1 Returned Structure

In a relation, once a tuple is selected, the tuple is returned. In XML, once an element (node) is selected, one may return not just the element, but also some structure around the element, most commonly all or a part of the sub-tree rooted at the element. How the data is stored and when the returned result is materialized can make a huge difference to query performance.

We consider the following cases:

1. Return only the element in question.
2. Return the entire sub-tree rooted at the element.
3. Return the element and all its immediate children.
4. Return the element and selected descendants.

To understand the role of returned structure in query performance, we use the query: Select all elements with aSixtyFour = 2. This query is run four times, corresponding to the four cases above. For the fourth case, only the descendants with aFour = 1 are returned. The remaining queries in the benchmark simply return the selected node (i.e. option 1 above), except when explicitly specified otherwise.

2.1.2 Simple Selection Queries

- **Exact Match Attribute Value Selection**
  - Selection based on the value of a string attribute.

    * **QS1.** High Selectivity: aString = "Sing a song of oneB1". Selectivity is approximately 1/16 (6.2%).
    * **QS2.** Low Selectivity: aString = "Sing a song of oneB4". Selectivity is approximately 1/128 (0.8%).
  - **QS3.** Selection based on the value of an integer attribute. We reproduce the same selectivities as in the string attribute case. The predicate aLevel = 13 has selectivity approximately 6.1%, and the predicate aLevel = 10 has selectivity approximately 0.8%.
  - **QS4.** Multi-attribute selection based on the value of two integer attributes. aSixteen = 1 and aFour = 1. Selectivity: 1.6% ((1/16) × (1/4) = 1/64).

- **Element Name Selection**

  QS5. Selection based on the element name, eOccasional. Selectivity: 1/64 (1.6%).

- **Element Content Selection**
  - **QS6.** Look for nodes that contains "oneB4". The likelihood of a single "PickWord" being from bucket 4 is 1/16, and given bucket 4, of being this particular word is 1/8. This gives a product probability of 1/128. (This string can also arise in bucket 16. But 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%).
  - **QS7.** Look for the keyword "oneB4" in the element content of OccasionalType nodes. The selectivity of this query is identical to the one above. The number of elements of the requisite type are only 1/64 in number.

- **Order-based Selection**
  - **QS8.** Select the second child of every node with aLevel = 7. Selectivity is 0.4%. (This is the same as the fraction of nodes at level 7, since each of these has more than two children).
  - **QS9.** Select the second child of every node with aLevel = 9. Selectivity is again 0.4%. The difference in performance between this and the query above is likely to be on account of fanout.

- **String Distance Selection**
  - **QS10.** Select all nodes with element content that has the keyword "oneB2" within four words of the keyword "Queen". Selectivity approximately 1/16 (6.2%). There are two occurrences of "PickWord" within four words of "Queen" and 14 occurrences that are further away. The probability of any one occurrence of "PickWord" being the selected keyword is 1/32 (3.1%).
  - **QS11.** Select all nodes with element content that has the keyword "oneB5" within four words of the keyword "Queen". Selectivity approximately 1/128 (0.8%). The probability of any one occurrence of "PickWord" being the selected keyword is 1/256 (0.4%).

2.1.3 Structural Selection Queries

All queries return the root of the selection pattern only, unless otherwise specified.

- **Parent-Child Selection Queries:**
  - **QS12.** Moderate selectivity of both parent and child. Select nodes with aLevel = 13 that have a child with attribute aSixteen = 3. The first predicate has selectivity of approximately 1/16. The child predicate also has 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%).
  - **QS13.** High selectivity of parent and low selectivity of child. Select nodes with aLevel = 15 that have a child with attribute aSixtyFour = 3. The first predicate has selectivity of approximately 1/4. The child predicate has a selectivity of 1/64. Following the same argument as above, the selectivity of the query as a whole is still 1/128 (0.8%).
  - **QS14.** Low selectivity of parent and high selectivity of child. Select nodes with aLevel = 11 that have a child with attribute aFour = 3. The first predicate has selectivity of approximately 1/64. The child predicate has a selectivity of 1/4. Following the same argument as above, the selectivity of the query as a whole is still 1/128 (0.8%).

- **Order-sensitive parent-child queries:**
  - **QS15.** Local Ordering. Select the second element with aFour = 1 below *each* element with aFour = 1. The selectivity of the latter predicate is 1/4. About half of these nodes

are at level 16, and have no children. Almost all the remaining nodes have exactly two children, each of which satisfies the former predicate with probability 1/4. Both children must satisfy this predicate for the second one to satisfy it, so the overall selectivity of this query is obtained by multiplying these probabilities: $(1/4) \times (1/2) \times (1/4) \times (1/4) = 1/128$ (0.8%).

- **QS16.** 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.
- **QS17.** Reverse Ordering. Select the last element with aSixteen = 1 below each element with aLevel = 13. 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 approximately 1/128 (0.8%).

• **Ancestor-Descendant Selection Queries:**

- **QS18.** Moderate selectivity of both ancestor and descendant. Select nodes with aLevel = 13 that have a descendant with attribute aSixteen = 3. The first predicate has selectivity of approximately 1/16. The second (descendant) predicate also has a selectivity of 1/16. Since each node at level 13 has fourteen descendants, there are 14 opportunities to satisfy the child predicate. Therefore the selectivity of the query as a whole is roughly $(1/16) \times (14/16) = 14/256$ (5.5%).
- **QS19.** High selectivity of ancestor and low selectivity of descendant. Select nodes with aLevel = 15 that have a descendant with attribute aSixtyFour = 3. The first predicate has selectivity of approximately 1/4. The second predicate has a selectivity of 1/64. The selectivity of the query as a whole is approximately 1/128. Note that there are only sixteen levels in the data tree. So a node at level 15 only has no descendants other than its own children and this query result is identical to the corresponding parent-child query above.
- **QS20.** Low selectivity of ancestor and high selectivity of descendant. Select nodes with aLevel = 11 that have a descendant with attribute aFour = 3. The first predicate has selectivity of approximately 1/64. The second predicate has a selectivity of 1/4 for one

descendant. Since each level 11 node has 72 descendants, finding such a descendant is virtually guaranteed, so that the selectivity of the query as a whole is about 1/64 (1.6%).

- **Ancestor Nesting in Ancestor-Descendant Query:** In the ancestor-descendant queries above, 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 the queries above.

  - **QS21.** Moderate selectivity of both ancestor and descendant. Select nodes with aSixteen = 3 that have a descendant with attribute aSixteen = 3. The first predicate has selectivity of approximately 1/16 (6.2%). The second (descendant) predicate also has a selectivity of 1/16 (6.2%).
  - **QS22.** High selectivity of ancestor and low selectivity of descendant. Select nodes with aFour = 3 that have a descendant with attribute aSixtyFour = 3. The first predicate has selectivity of approximately 1/4 (25%). The second predicate has a selectivity of 1/64 (1.6%).
  - **QS23.** Low selectivity of ancestor and high selectivity of descendant. Select nodes with aSixtyFour = 3 that have a descendant with attribute aFour = 3. The first predicate has selectivity of approximately 1/64 (1.6%). The second predicate has a selectivity of 1/4 (25%).

  The overall selectivities of these queries cannot be the same as that of the "equivalent" unnested queries on two accounts – 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 forces work in opposite directions, but may not quite cancel each other out in any specific case. We have chosen to 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:**
  QS24. For the last query above, return not just the root of the selection pattern, but also the descendant node. Thus the returned structure is a pair of nodes with an inclusion relationship between them.

- **Complex Pattern Selection Queries:**

  - **QS25.** One chain query with three parent-child joins with the selectivity pattern: high-low-low-high, 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.
  - **QS26.** One twig query with two parent child selection, low selectivity of parent aLevel=11, high selectivity of left child aFour=3 and low selectivity of right child aSixtyFour=3.
  - **QS27.** One twig query with two parent child selection, high selectivity of parent aFour=1, low selectivity of left child aLevel=11 and low selectivity of right child aSixtyFour=3.
  - **QS28.** Repeat the above three queries using ancestor-descendant in place of parent-child.
  - **QS29.** Repeat last query, but use ancestor-descendant for the right child and parent-child for the left child in the twig query.

2.2 Value Join Queries

- **QJ1.** Low selectivity join: select nodes based on aSixtyFour =3 and join with nodes with aSixtyFour = 4 based on equality of the aLevel attribute. 1/64 of the nodes satisfy either selection condition, independent of node level. Since the conditions are independent, the join selectivity is simply the product $(1/64) \times (1/64) = 1/4096(0.02\%)$.

- **QJ2.** High selectivity join: select nodes based on aFour =3 and join with nodes with aFour = 4 based on equality of the aLevel attribute. 1/4 of the nodes satisfy either selection condition, independent of node level. The join selectivity is 1/16(6.2%).

In computing the value-based joins above, one would naturally expect *both* nodes participating in the join to be returned. As such, the return structure expected is a tree per join-pair. Each tree has a join-node as root, and two children, one corresponding to each element participating in the join.

2.3 Pointer-based Join Query

- **QJ3.** Low selectivity join: select all eOccasional nodes that point to a node with aSixtyFour =3. Selects 1/64 (1.6%) of all the eOccasional nodes.
- **QJ4.** High selectivity join: select all eOccasional nodes that point to a node with aFour =3. Selects 1/4 (25%) of all the eOccasional nodes.

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

2.4 Aggregate Queries

- **QA1. Value Aggregation.** Over all nodes at level 15, compute the average value for the aSixtyFour attribute. Recall that about 1/4 of the nodes are at level 15 and will participate in this aggregation.
- **QA2. Value Aggregation with Groupby.** Group nodes by level. Over all nodes at each level, compute the average value of the aSixtyFour attribute. The return structure is a tree, with a dummy root and a child for each group. Each lead (child) node has one attribute for the level and one attribute for the average value.
- **QA3. 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. 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.
- **QA4. Value Aggregate Selection.** Select elements that have at least two occurrences of the keyword oneB1 in their content. There are sixteen chances for this keyword to appear in the content of each element, and the probability of its being chosen is 1/16. Solving the binomial probability distribution that results, there is about a 1/4 chance of its occurring at least twice. So the selectivity of this query is 1/4.
- **QA5. Structural Aggregate Selection.** 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 obtained by multiplying these probabilities: $1/2 \times 1/4 \times 1/4 = 1/32(3.1\%)$.

- **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 root, with 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 256 nodes at level 7, irrespective of the database scaling. Under each of these, the sub-tree goes to level 16, and so is exactly ten levels high, again irrespective of scaling. However, determining this height may require exploring substantial parts of the database.

2.5 Miscellaneous Queries

These are additional important queries to cover all performance critical aspects of XML Processing [1].

- **QM1. Missing Elements.** Find all BaseType elements that there is no OccasionalType elements below them.

- **QM2. Casting.** Retrieve the integer values of **aUnique2** attribute of elements that have aLevel = 13 which has selectivity approximately 6.1%, and the predicate aLevel = 10 which has selectivity approximately 0.8%.

2.6 Updates

- **QU1. Point Inserts.** Insert a new node below the node with aUnique1 = 10102.

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

- **QU3. Bulk Inserts.** Insert a new 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, not necessarily contiguous with the values already assigned in the database.

- **QU4. Bulk Deletes.** 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 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.

References

[1] 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. `http://www.acm.org/sigmod/record/issues/0109/index.html`.

[2] D. C. Fallside (eds.). XML Schema Part 0: Primer, May 2001. `http://www.w3.org/TR/xmlschema-0/`.