# The Michigan Benchmark: Towards XML Query Performance Diagnostics

Kanda Runapongsa    Jignesh M. Patel    H. V. Jagadish    Yun Chen    Shurug Al-Khalifa
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122 USA
{krunapon, jignesh, jag, yunc, shurug}@eecs.umich.edu

**Abstract**

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

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

## 1 Introduction

XML query processing has taken on considerable importance recently, and several XML databases [3, 9–11, 13, 25, 27] 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, 20, 22]. 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 benchmark for XML DBMSs. The benchmark that we propose to achieve this goal is called the Michigan benchmark.

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

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

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

The main contributions of this paper are:

- The identification of XML data characteristics that may impact the performance of XML query processing engines.
- A single heterogeneous data set against which carefully specified queries can be used to evaluate system performance for XML data with various characteristics.
- Insights from running this benchmark on three database systems: a commercial native XML database system, a native XML database system that we have been developing at the University of Michigan, and a commercial object-relational DBMS.

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

## 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, Barbosa et al. [6] proposed a template-based data generator for XML, ToXgene, which can generate multiple tunable data sets. The ToXgene user can specify the distribution of different element values in these data sets. In contrast to these previous data generators, the data generator in this proposed benchmark produces an XML data set designed to test different XML data characteristics that may affect the performance of XML engines. In addition, the data generator requires only a few parameters to vary the scalability of the data set. The schema of the data set is also available to exploit.

Four benchmarks [5, 7, 20, 22] have been proposed for evaluating the performance of XML data management systems. XMach-1 [7] and XMark [22] 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 [22], the data is based on an Internet auction application that consists of relatively structured and data-oriented parts. XOO7 [20] is an XML version of the OO7 Benchmark [18], which is a benchmark for OODBMSs. The OO7 schema and instances are mapped into a Document Type Definition (DTD), and the eight OO7 queries are translated into three respective languages for query processing engines: Lore [15, 19], Kweelt [21], and an ORDBMS. Recognizing that different applications requires different benchmarks, Yao et al. [5] have recently proposed, Xbench, which is a family of a number of different application benchmarks.

While each of these benchmarks provides an excellent measure of how a test system would perform against data and queries in their targeted XML application, it is difficult to extrapolate the results to data sets and queries that are different from ones in the targeted domain. Although the queries in these benchmarks are designed to test different performance aspects of XML engines, they cannot be used to perceive the system performance change as the XML data characteristics change. On the other hand, we have different queries to analyze the system performance with respect to different XML data characteristics, such as tree fanout and tree depth; and different query characteristics, such as predicate selectivity.

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

## 3 Benchmark Data Set

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

### 3.1 A Discussion of the Data Characteristics

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

#### 3.1.1 Depth and Fanout

Depth and fanout are two structural parameters important to tree-structured data. The depth of the data tree can have a significant performance impact, for instance, when computing indirect containment relationships between ancestor and descendant nodes in the tree. Similarly, the fanout of nodes 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 of a node).

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

We create a base benchmark data set of a depth of 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. Similarly, we specify high (13) and low (2) fanouts at different levels of the tree as shown in Figure 1. The fanout of 1/13 at level 8 means that every thirteenth node at this level has a single child, and all other nodes are childless leaves. This variation in fanout is designed to permit queries that focus isolating the fanout factor. For

| 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

instance, the number of nodes is the same (2,704) at levels 7 and 9. Nodes at level 7 have a fanout of 13, whereas nodes at level 9 have a fanout of 2. A pair of queries, one against each of these two levels, can be used to isolate the impact of fanout. In the rightmost column of Figure 1, "% of Nodes" is the percentage of the number of nodes at each level to the number of total nodes in a document.

### 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, depths, or fanouts. We would like to isolate the effect of the number of nodes from 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 numbers of fanouts of nodes at only a few levels, namely levels 5-8. 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.

We propose to scale the Michigan benchmark in discrete steps. The default data set, called **DSx1**, has 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 72,351K respectively [1].

## 3.2 Schema of Benchmark Data

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

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

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

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

The XML Schema specification of the benchmark data set is shown in Figure 2.

## 3.3 String Attributes and Element Content

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

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

[1] this translates into a scale factor of 9.9x and 99.4x.

[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:element name="eNest" type="BaseType">
  <xsd:complexType name="BaseType" mixed="true">
  <xsd:sequence>
    <xsd:element name="eNest" type="BaseType" minOccurs="0" maxOccurs="unbounded">
      <xsd:key name="aU1PK">
        <xsd:selector xpath=".//eNest"/>
        <xsd:field xpath="@aUnique1"/>
      </xsd:key>
      <xsd:unique name="aU2">
        <xsd:selector xpath=".//eNest"/>
        <xsd:field xpath="@aUnique2"/>
      </xsd:unique>
    </xsd:element>
    <xsd:element name="eOccasional" type="OccasionalType" minOccurs="0">
      <xsd:keyref name="aU1FK" refer="aU1PK">
        <xsd:selector xpath="."/>
        <xsd:field xpath="@aRef"/>
      </xsd:keyref>
    </xsd:element>
  </xsd:sequence>
  <xsd:attributeGroup ref="BaseTypeAttrs"/>
  </xsd:complexType>
  <xsd:complexType name="OccasionalType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="aRef" type="xsd:integer" use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:attributeGroup name="BaseTypeAttrs">
    <xsd:attribute name="aUnique1" type="xsd:integer" use="required"/>
    <xsd:attribute name="aUnique2" type="xsd:integer" use="required"/>
    <xsd:attribute name="aLevel" type="xsd:integer" use="required"/>
    <xsd:attribute name="aFour" type="xsd:integer" use="required"/>
    <xsd:attribute name="aSixteen" type="xsd:integer" use="required"/>
    <xsd:attribute name="aSixtyFour" type="xsd:integer" use="required"/>
    <xsd:attribute name="aString" type="xsd:string" use="required"/>
  </xsd:attributeGroup>
</xsd:schema>
```

Figure 2: Benchmark Specification in XML Schema

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.

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

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

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

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

Figure 3: Generation of the String Element Content

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

## 4 Benchmark Queries

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

We find it useful to refer to simple 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 the benchmark queries in detail. In these query descriptions, the types of the nodes are assumed to be BaseType 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 bibliographic database, that seeks books, published in the year 2002, by an author with name including the string "Blake". This apparently straightforward selection query involves matches in the database to a 4-node "query pattern", with predicates associated with each of these four (namely book, year, author, and name). Once a match has been found for this pattern, we may be interested in returning only the book element, all the nodes that participated in the match, or various other possibilities. We attempt to organize the various sources of complexity in the following.

### 4.1.1 Returned Structure

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

To understand the role of returned structure in query performance, we use the query, "Select all elements with aSixtyFour = 2." The selectivity of this query is 1/64 (1.6%)[3]

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

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

### 4.1.2 Simple Selection

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

- **Exact Match Attribute Value Selection**
  *Value-based selection on 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%.

  *Value-based selection on an integer attribute.*
  These following queries have almost the same selectivities as the above string attribute queries.
  **QS3. Low selectivity.** Select nodes with aLevel = 10. Selectivity is 0.7%.

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

  *Selection on range values.*
  **QS5.** Select nodes with aSixtyFour between 5 and 8. Selectivity is 6.3%.

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

  *Multiple-attribute selection.*
  **QS7.** Select nodes with attributes aSixteen = 1 and aFour = 1. Selectivity is 1.6%.

- **Element Name Selection**
  **QS8.** Select nodes with the element name eOccasional. Selectivity is 1.6%.

- **Order-based Selection**
  **QS9. High fanout.** Select the second child of every node with aLevel = 7. Selectivity is 0.4%.

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

[3]Detailed computation of the query selectivities can be found in Appendix A.

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

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

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

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

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

### 4.1.3 Structural Selection

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  **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.

The overall selectivities of these queries (QS24-QS26) cannot be the same as that of the "equivalent" unnested queries (QS21-QS23) for two situations – first, the same descendants can now have multiple ancestors they match, and second, the number of candidate descendants is different (fewer) since the ancestor predicate can be satisfied by nodes at any level (and will predominantly be satisfied by nodes at levels 15 and 16, due to their large numbers). These two effects may not necessarily 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).

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

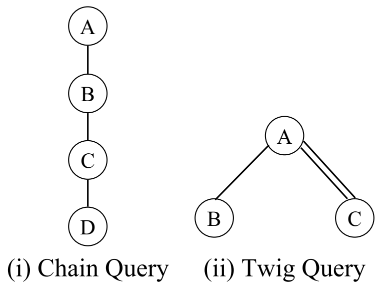

Figure 4: Samples of Chain and Twig Queries

The benchmark uses the following complex queries:

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

  **QS29. One twig query with two parent-child joins with the selectivity pattern: low-high, low-low.**

Select parent nodes with aLevel = 11 (sel=1.5%) that have a child with aFour = 3 (sel=1/4), and another child with aSixtyFour = 3 (sel=1/64).

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

- **Ancestor-Descendant Complex Pattern Selection**
  **QS31-QS33.** Repeat queries QS28-QS30, but using ancestor-descendant in place of parent-child.

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

- **Negated Selection**
  In XML, some elements are optional and some queries test the existence of these optional elements. Negated selection query selects elements which does not contain a descendant that is an optional element.

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

## 4.2 Value-Based Join

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

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

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

## 4.3 Pointer-Based Join

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

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

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

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

## 4.4 Aggregation and Update Queries

The benchmark also contains three value-based aggregate queries, four structure-based aggregate queries, and seven update queries, which include point update and delete, bulk insert and delete, bulk load, bulk reconstruction, and bulk restructuring.

## 4.5 Aggregation

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

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

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

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

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

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

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

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

### 4.6 Update

The benchmark also contains seven update queries, which include point update and delete, bulk insert and delete, bulk load, bulk reconstruction, and bulk restructuring.

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

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

**QU3. Bulk Insert.** 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 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 OccasionalType.

**QU7. Restructuring.** For a node $u$ of type OccasionalType, 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 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 commercial XML DBMS, a university native XML DBMS, and one leading commercial ORDBMS. Due to the nature of the licensing agreement for the commercial systems, we can not disclose the actual names of the system, and will refer to the commercial native system as **CNX**, and the commercial ORDBMS as **COR**.

The native XML database is Timber [27], a university native XML DBMS system that we are developing at the University of Michigan [27]. Timber uses the Shore storage manager [8], and implements various join algorithms, query size estimation, and query optimization techniques that have been developed for XML DBMSs.

The COR is provided by a leading database vendor, and we used the Hybrid inlining algorithm to map the data into a relational schema [24]. To generate good SQL queries, we adopted the algorithm presented in [14]. The queries in the benchmark were converted into SQL queries (sanitized to remove any system-specific keywords in the query language) which can be found in the Appendix B.

The commercial native XML system provides an XPath interface and a recently released XQuery interface. We started by writing the benchmark queries in XQuery. However, we found that the XQuery interface was unstable and in most cases would hang up either the server or the Java client, or run out of memory on the machine. Unfortunately, no other interface is available for posing XQueries to this commercial system. Consequently, we reverted to writing the queries using XPath expression, which implies that join queries are written as nested XPath expressions, that get evaluated using a nested-loops paradigm. In all the cases that we could run queries using the XQuery interface, the XPath approach was faster. Consequently, all the numbers reported here are written as XPath queries. The actual queries for the commercial native XML system (sanitized to remove any system-specific keywords in the query language) can be found in the Appendix C.

## 5.1 Experimental Platform and Methodology

All experiments were run on a single-processor 550 MHz Pentium III machine with 256 MB of main memory. The benchmark machine was running the Windows2000 and was configured with a 20 GB IDE disk. All three systems were configured to use a 64 MB buffer pool size.

### 5.1.1 System Setup

For both commercial systems, we used default settings that the system chooses during the software installation. The only setting that we changed for COR was to enable the use of hash joins, as we found that query response times generally improved with this option turned on. For COR, after loading the data we update all statistics to provide the optimizer with the most current statistical information.

For CNX, we tried running the queries with and without indices. CNX permits building both structure (i.e., path indices), and value indices. Surprisingly, we found that indexing did not help the performance of the benchmark queries, and in fact in most cases actually reduced the performance of the queries. In very few cases, the performance improved but by less than 20% over the non-indexed case. The reason for the ineffectiveness of the index is that CNX indexing can not handle the "//" operator, which is used often in the benchmark. Furthermore, the index is not effective on BaseType elements as it is recursively nested below other BaseType elements.

### 5.1.2 Data Sets

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

For the purpose of the experiment, we loaded and wrote queries against the scaled down set before using the base data set. The smaller data set size reduced the time to set up the queries and load the scripts, for all systems. The same queries and scripts were then reused for the base data set. Since we expect that this strategy may also be useful to other users of this benchmark, the data generator for this benchmark, which is available for free download from the benchmark's website [26], allows generation of this scaled down data set.

### 5.1.3 Measurements

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

### 5.1.4 Benchmark Results

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

| Discussed Section | Query Group (Queries in Group) | Geometric Mean Response Times (seconds) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | ds0.1x | | | | ds1x | | | |
| | | CNX | | Timber | COR | CNX | | Timber | COR |
| | | Idx | No Idx | | | Idx | No Idx | | |
| 5.2.1 | Returned structure (QR1-QR4) | 2.95 | 2.53 | 0.06 | 0.27 | 10.35 | 9.40 | 0.37 | 3.87 |
| 5.2.2 | Exact match attribute value selection (QS1-QS7) | 2.64 | 2.25 | 0.04 | 0.04 | 7.61 | 6.73 | 0.48 | 0.28 |
| 5.2.3 | Element name selection (QS8) | 2.72 | 2.12 | 0.01 | 0.02 | 7.21 | 5.98 | 0.08 | 0.15 |
| 5.2.4 | Order-based selection (QS9-QS10) | 2.53 | 2.18 | 0.00 | 0.06 | 7.25 | 6.25 | 0.05 | 0.61 |
| 5.2.5 | Element content selection (QS11-QS12) | 2.84 | 2.47 | 0.07 | 0.12 | 8.23 | 7.01 | 0.69 | 2.75 |
| 5.2.5 | String distance selection (QS13-QS14) | N/A | N/A | 3.00 | 1.12 | N/A | N/A | 32.92 | 39.52 |
| 5.2.6 | Order-sensitive selection (QS15-QS17) | 2.65 | 2.28 | 0.02 | 0.08 | 7.76 | 6.75 | 0.54 | 0.26 |
| 5.2.7 | Parent-child (P-C) selection (QS18-QS20) | 2.72 | 2.36 | 0.13 | 0.04 | 7.65 | 6.89 | 1.38 | 0.34 |
| 5.2.7 | Ancestor-descendant (A-D) selection (QS21-QS23) | 3.21 | 2.68 | 0.23 | 2.14 | 8.76 | 7.74 | 1.41 | 17.92 |
| 5.2.7 | Ancestor nesting in A-D selection (QS24-QS26) | 3.31 | 2.76 | 0.13 | 1.16 | 8.96 | 8.02 | 1.39 | 12.65 |
| 5.2.8 | P-C complex pattern selection (QS27-QS30) | 3.85 | 3.26 | 0.25 | 0.03 | 8.19 | 7.31 | 2.56 | 0.50 |
| 5.2.8 | A-D complex pattern selection (QS31-QS34) | 6.13 | 3.64 | 0.27 | 2.49 | 11.60 | 10.63 | 2.75 | 24.74 |
| 5.2.9 | Negated selection (QS35) | 3.19 | 2.84 | 1.29 | 2.10 | 82.06 | 66.15 | 12.58 | 23.38 |
| 5.2.10 | Value-based join (QJ1-QJ2) | 359.14 | 359.17 | 1.72 | 0.05 | 2497.50 | 2537.92 | 18.82 | 0.42 |
| 5.2.10 | Pointer-based join (QJ3-QJ4) | 163.55 | 161.80 | 3.15 | 0.02 | 1330.69 | 1339.54 | 19.73 | 0.14 |
| 5.2.11 | Value aggregation (QA1-QA3) | N/A | N/A | 11.07 | 0.23 | N/A | N/A | 1184.69 | 2.31 |
| 5.2.11 | Structural aggregation (QA4-QA6) | N/A | N/A | 0.36 | 1.06 | N/A | N/A | 209.84 | 11.97 |
| 5.2.12 | Update (QU1-QU7) | N/A | N/A | N/A | 2.71 | N/A | N/A | N/A | 78.18 |

Figure 5: Benchmark Numbers for Three DBMSs. CNX - a commercial native XML DBMS, Timber - a university native XML DBMS, and COR - a commercial ORDBMS. N/A indicates that the queries could not be run on that system.

From Figure 5, we observe that Timber is very efficient at processing XML structural queries. The implementation of "traditional" relational-style queries such as value-based joins is not highly tuned in Timber. This is primarily because Timber is a research prototype and most of the development attention has been paid to the XML query processing issues that are not covered by traditional relational techniques.

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

From Figure 5, we observe that CNX is usually slower than the other two systems. A large query overhead, of about 2 secs, is incurred by CNX, even for very small queries. CNX is considerably slower than Timber, and faster than COR only on the ancestor-descendant queries.

We suspect that the reason for this to happen is fourfold. First, although we had structure indexes built in the schemas of the database, most of the queries involve the "//" operator, which incurs post-processing on the server that dominates the major chunk of the query processing time. Second, we suspect that CNX only builds tag indexes for its structural indexes. As a result, since all elements in the benchmark data set are either eNest elements or eOccasional elements, tag indexes are essentially useless. Furthermore, when the system is forced to follow the tag indexes, random accesses occur and it results in compromised performance. Third, the value indexes set on the attributes of the eNest nodes may cause random disk access at query time if not all the documents can fit into memory, in which we naturally incur a much higher disk access overhead than in the non-indexed data sets where we simply do a sequential scan on the disk to retrieve the matching nodes. Finally, in the above situation, the optimizer should have been able to evaluate different query plans and choose the optimal plan, which in this case should be the sequential scan plan, while obviously it was not the case in our experiments.

## 5.2 Detailed Performance Analysis

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

### 5.2.1 Returned Structure (QR1-QR4)

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

| Query | Query Description | Sel.(%) | Response Times (seconds) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | ds0.1x | | | | ds1x | | | |
| | | | CNX | | Timber | COR | CNX | | Timber | COR |
| | | | Idx | No Idx | | | Idx | No Idx | | |
| QR1 | Return result element | 1.6 | 2.52 | 2.18 | 0.01 | 0.02 | 7.43 | 6.19 | 0.08 | 0.16 |
| QR2 | Return element and immediate children | 1.6 | 2.03 | 1.68 | 0.02 | 0.31 | 5.96 | 4.83 | 0.27 | 2.59 |
| QR3 | Return entire sub-tree | 1.6 | 3.97 | 3.63 | 0.26 | 1.09 | 11.36 | 10.17 | 387.23 | 26.09 |
| QR4 | Return element and selected descendants | 1.6 | 2.73 | 2.37 | 0.19 | 0.97 | 8.31 | 7.14 | 2.44 | 20.57 |

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

Examining the performance of the returned structure queries, QR1-QR4 in Figure 6, we observe that the returned structure has an impact on all systems. Timber performs the worst when the whole subtree is returned (QR3). This is surprising since Timber stores elements in depth-first order, so that retrieving a sub-tree should be a fast sequential scan. It turns out that Timber uses SHORE [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 to begin taking steps to address it.

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

CNX produces results in reasonably short times. Surprisingly, returning the result element itself (QR1) takes a little more time than returning the element and its immediate children (QR2). We suspect that CNX returns the entire subtree by default, and then will carry out post-processing selection to return the element itself, which takes more time. This also explains why CNX incurs more query processing time than the other DBMSs in most queries.

## Simple Selection (QS1-QS10)

In this section, we examine the performance of the three systems for the simple selection queries. The performance numbers are shown in Figure 7.

| Query | Query Description | Sel.(%) | Response Times (seconds) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | ds0.1x | | | | ds1x | | | |
| | | | CNX | | Timber | COR | CNX | | Timber | COR |
| | | | Idx | No Idx | | | Idx | No Idx | | |
| QS1 | Selection on string attribute value (low sel.) | 0.8 | 2.36 | 1.99 | 0.03 | 0.02 | 6.96 | 6.06 | 0.05 | 0.08 |
| QS2 | Selection on string attribute value (high sel.) | 6.3 | 2.40 | 2.05 | 0.03 | 0.06 | 7.08 | 6.21 | 0.34 | 0.63 |
| QS3 | Selection on integer attribute value (low sel.) | 0.7 | 2.62 | 2.25 | 0.01 | 0.02 | 7.71 | 6.76 | 0.04 | 0.08 |
| QS4 | Selection on integer attribute value (high sel.) | 6.0 | 2.64 | 2.28 | 0.03 | 0.06 | 7.75 | 6.82 | 0.30 | 0.57 |
| QS5 | Selection on range values | 6.3 | 3.21 | 2.81 | 0.03 | 0.06 | 9.24 | 8.23 | 0.23 | 0.60 |
| QS6 | Selection with sorting | 6.0 | 2.72 | N/A | 1.83 | 0.06 | 7.53 | N/A | 71.71 | 0.58 |
| QS7 | Multiple-attribute selection | 1.6 | 2.60 | 2.23 | 0.16 | 0.02 | 7.25 | 6.50 | 1.70 | 0.17 |
| QS8 | Element name selection | 1.6 | 2.72 | 2.21 | 0.01 | 0.02 | 7.21 | 5.98 | 0.08 | 0.15 |
| QS9 | Order-based selection (high fanout) | 0.4 | 2.53 | 2.19 | 0.003 | 0.06 | 7.71 | 6.31 | 0.05 | 0.61 |
| QS10 | Order-based selection (low fanout) | 0.4 | 2.52 | 2.18 | 0.003 | 0.06 | 7.43 | 6.19 | 0.06 | 0.62 |

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

### 5.2.2 Exact Attribute Value Selection (QS1-QS7)

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

Overall, CNX does not perform as well as the other two DBMSs. It is interesting to notice that although selectivity does have some impact on CNX, it is not as strong as on the other two DBMSs (this is true even with indexing in CNX). Although the response times of the high selectivity queries (QS2, QS4) are higher than the response times of the low selectivity queries (QS1, QS3), the difference does not reflect a linear growth. Selection on short strings takes a little more time than that on integers, although the difference is negligible.

**Range Selection (QS5)** Both Timber and COR systems handle a range predicate just as well as an equality predicate. In both systems, the performance of the range predicate query (QS5) is almost the same as that of the comparable equality selection queries (QS2 and QS4). On the other hand, CNX takes a little more time to evaluate the range predicate query than the comparable equality selection queries.

**Multiple-attribute Selection and Sorting (QS6-QS7)** Currently, Timber does not support multiple-attribute indices. This is why it has high response times for QS6 and QS7 than for QS3. To evaluate QS6, Timber needs to use an unclustered index to access all nodes that satisfy the predicate aLevel = 13; then it has to retrieve the actual nodes and sort these nodes on the value of the aSixtyFour attribute. To evaluate QS7, Timber requires two index accesses (one for each predicate) and a set intersection between them. On the other hand, the COR performs well on QS6 and QS7, since it uses the appropriate multiple-attribute indices. CNX does not perform as well as other databases. Note that since the data sets were loaded in small partitions, it made sorting impossible for COR on data without index.

### 5.2.3 Element Name Selection (QS8)

Both Timber and COR resolve the query request for a given element name very well. This is because Timber uses an index on element names, and COR simply requests all tuples from the table corresponding to the given element name. Like other selection queries, CNX does not perform as well as the other databases.

### 5.2.4 Fanout (QS9-QS10)

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

### 5.2.5 Text Processing (QS11-QS14)

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

| Query | Query Description | Sel.(%) | Response Times (seconds) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | ds0.1x | | | | ds1x | | | |
| | | | CNX | | Timber | COR | CNX | | Timber | COR |
| | | | Idx | No Idx | | | Idx | No Idx | | |
| QS11 | Element content selection (low sel.) | 0.2 | 2.03 | 1.68 | 0.08 | 0.02 | 5.96 | 4.83 | 0.78 | 0.17 |
| QS12 | Element content selection (high sel.) | 12.5 | 3.97 | 3.63 | 0.06 | 0.97 | 11.36 | 10.17 | 0.61 | 43.85 |
| QS13 | String distance selection (low sel.) | 0.8 | N/A | N/A | 2.49 | 0.95 | N/A | N/A | 27.23 | 42.79 |
| QS14 | String distance selection (high sel.) | 6.3 | N/A | N/A | 3.61 | 1.31 | N/A | N/A | 39.79 | 45.42 |

Figure 8: Benchmark Numbers for Three DBMSs on Simple Selection Queries

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

CNX supports element content selection (QS11 and QS12), but does not support string distance selection yet (QS13 and QS14). Although processing long strings (QS11 and QS12) is supposed to be more expensive than processing short ones (QS1 and QS2), the query performance of CNX on the former two queries is not affected much in this case. Two other points are worth noticing: selectivity does not affect much the performance of CNX; CNX seems to scale up to the database size better than COR and Timber, especially concerning high selectivity queries (QS12).

## Structural Selection (QS15-QS35)

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

The performance of the order selection queries, QS15-QS17, is shown in Figure 9.

| Query | Query Description | Sel.(%) | Response Times (seconds) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | ds0.1x | | | | ds1x | | | |
| | | | CNX | | Timber | COR | CNX | | Timber | COR |
| | | | Idx | No Idx | | | Idx | No Idx | | |
| QS15 | Local ordering | 3.1 | 2.73 | 2.37 | 1.45 | 0.08 | 8.31 | 7.14 | 14.58 | 1.02 |
| QS16 | Global ordering | 1 node | 2.56 | 2.19 | 0.00 | 0.01 | 7.32 | 6.39 | 0.01 | 0.03 |
| QS17 | Reverse ordering | 0.7 | 2.65 | 2.29 | 0.08 | 0.07 | 7.67 | 6.75 | 1.06 | 0.55 |

Figure 9: Benchmark Numbers for Three DBMSs on Simple Selection Queries

In CNX, local ordering (QS15), global ordering (QS16), and reverse ordering (QS17) are not so much different from each other.

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

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

### 5.2.7 Simple Containment Selection (QS18-QS26)

Figure 10 shows the performance of selected structural selection queries, QS18-QS26, and QS35. In this figure, “P-C:low-high” refers to the join between a parent with low selectivity and a child with high selectivity, whereas, “A-D:high-low” refers to the join between an ancestor with high selectivity and a descendant with low selectivity.

As seen from the results for the direct containment queries (QS18-QS20) in Figure 10, COR processes direct containment queries better than Timber, but Timber handles indirect containment queries (QS21-QS26) better.

CNX underperforms as compared to the other systems on direct containment queries (QS18-QS20). However, on indirect containment queries (QS21-QS26), it often performs better than COR. CNX only has slightly performance on direct containment queries (QS18-QS20) than on indirect containment queries (QS21-QS27). Examining the effect of query selectivities on CNX query execution (see QS21-QS23 as an example), we notice that the execution times are relatively immune to the query selectivities, implying that the system does not effectively exploit the differences in query selectivities in picking query plans.

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

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

| Query | Query Description | Sel.(%) | Response Times (seconds) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | ds0.1x | | | | ds1x | | | |
| | | | CNX | | Timber | COR | CNX | | Timber | COR |
| | | | Idx | No Idx | | | Idx | No Idx | | |
| QS18 | P-C: medium-medium | 0.7 | 2.67 | 2.30 | 0.08 | 0.02 | 7.55 | 6.73 | 0.85 | 0.25 |
| QS19 | P-C: high-low | 0.7 | 2.92 | 2.55 | 0.17 | 0.05 | 8.10 | 7.40 | 1.79 | 0.44 |
| QS20 | P-C: low-high | 0.7 | 2.59 | 2.23 | 0.17 | 0.05 | 7.32 | 6.58 | 1.73 | 0.34 |
| QS21 | A-D:medium-medium | 3.5 | 3.26 | 2.73 | 0.09 | 2.22 | 8.79 | 7.77 | 0.93 | 20.15 |
| QS22 | A-D:high-low | 0.7 | 3.11 | 2.57 | 0.18 | 0.94 | 8.44 | 7.40 | 1.72 | 5.64 |
| QS23 | A-D:low-high | 1.5 | 3.26 | 2.73 | 0.16 | 4.69 | 9.07 | 8.06 | 1.74 | 50.65 |
| QS24 | Ancestor nesting in A-D:medium-medium | 1.0 | 3.10 | 2.56 | 0.08 | 2.16 | 8.27 | 7.32 | 0.87 | 20.03 |
| QS25 | Ancestor nesting in A-D:high-low | 1.7 | 4.22 | 3.68 | 0.19 | 0.95 | 11.44 | 10.44 | 1.94 | 8.83 |
| QS26 | Ancestor nesting in A-D:low-high | 0.5 | 2.77 | 2.23 | 0.15 | 0.92 | 7.59 | 6.74 | 1.61 | 11.73 |
| QS27 | Ancestor nesting in A-D:low-high (a pair of nodes returned) | 5.0 | 3.77 | 2.95 | 0.16 | 0.97 | 9.81 | 8.67 | 1.76 | 12.35 |
| QS28 | P-C chain:high-low-low-high | 0.0 | 2.75 | 2.39 | 0.57 | 0.06 | 7.98 | 7.01 | 10.61 | 0.77 |
| QS29 | P-C twig:low-high, low-low | 0.0 | 2.61 | 2.24 | 0.19 | 0.02 | 7.45 | 6.71 | 3.57 | 0.48 |
| QS30 | P-C twig:high-low, high-low | 0.0 | 8.10 | 7.14 | 0.21 | 0.02 | 7.73 | 6.98 | 3.89 | 0.34 |
| QS31 | A-D chain:high-low-low-high | 0.4 | 18.00 | 4.04 | 0.56 | 20.22 | 16.20 | 16.59 | 10.56 | 190.19 |
| QS32 | A-D twig:low-high, low-low | 0.9 | 4.16 | 3.29 | 0.20 | 2.64 | 10.64 | 9.56 | 3.67 | 33.04 |
| QS33 | A-D twig:high-low, high-low | 0.4 | 6.22 | 5.35 | 0.21 | 1.24 | 12.49 | 11.21 | 3.84 | 10.86 |
| QS34 | Twig with one P-C (high-low) and one A-D (high-low) | 0.2 | 3.04 | 2.47 | 0.21 | 0.58 | 8.43 | 7.16 | 3.90 | 5.49 |
| QS35 | Negated selection | 93.2 | 3.19 | 2.84 | 1.29 | 2.10 | 82.06 | 66.15 | 12.58 | 23.38 |

Figure 10: Benchmark Numbers for Three DBMSs on Simple Selection Queries

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

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

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

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

### 5.2.8 Complex Pattern Containment Selection (QS28-QS34)

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

COR takes much more time to answer ancestor-descendant chain joins (QS31-QS33) than parent-child chain joins (QS28-QS30). Again, the high response times of ancestor-descendant queries are due to the recursive SQL, which is expensive to compute. In constructing the SQL queries for the COR, we followed the techniques described in [14] for choosing the order of the joins; as expected choosing a good join order resulted in a substantial reduction in performance in some case.

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

CNX does not perform as well as Timber and COR. Like Timber, CNX also breaks the complex pattern queries into a series of binary containment joins. Unlike Timber, indirect containment joins in CNX take more time than direct containment joins over all, for which the reason may be inefficient implementation of structural join algorithms in CNX. Notice that for QS31, the indexed version of the ds0.1x data set takes considerably more than the non-indexed version. This might be because the inefficient implementation of the structural indexing in CNX caused the system to chase each level of nesting in order to find the nodes in question. The index access is likely to take more than the sequential scan that occurs in the non-indexed database. This is not the case for the ds1x data set because both the index access and the sequential scan probably take the same amount of time.

### 5.2.9 Irregular Structure (QS35)

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

While looking for irregular data structures, CNX performs reasonably well on the small scale database, but as one might notice, it does not scale very well like with other queries. The selectivity of this query if fairly high (93.2%), and as the database size increases, the return result grows dramatically. CNX seems to spends a large part of its execution time in processing the results at the client, and this part does not seem to scale very well.

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 COR, there are two ways to implement this query. A naive way is to use a set difference operation which results in a very long response time (1517.4 seconds for the ds1x data set). This long response time is because COR first needs to find a set of elements that contain the missing elements (using a recursive SQL query), and then find elements that are not in that set. The second alternative of implementing this query is to use a left outer join. That is first create a view that selects all BaseType elements have some OccasionalType descendants (this requires a recursive SQL statement). Then compute a left-outer join between the view and the relation that holds

| Query | Query Description | Sel.(%) | Response Times (seconds) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | ds0.1x | | | | ds1x | | | |
| | | | CNX | | Timber | COR | CNX | | Timber | COR |
| | | | Idx | No Idx | | | Idx | No Idx | | |
| QJ1 | Value-based join (low sel.) | 1.6 | 188.88 | 187.5 | 0.69 | 0.03 | 1247.59 | 1268.4 | 18.82 | 0.21 |
| QJ2 | Value-based join (high sel.) | 6.3 | 682.87 | 687.9 | 4.29 | 0.08 | $>1$ hr | $>1$ hr | $>1$ hr | 0.83 |
| QJ3 | Pointer-based join (low sel.) | 0.02 | 161.50 | 160.09 | 0.73 | 0.01 | 1307.6 | 1320.52 | 20.08 | 0.05 |
| QJ4 | Pointer-based join (high sel.) | 0.4 | 165.52 | 163.5 | 13.63 | 0.05 | 1354.2 | 1358.83 | 19.38 | 0.42 |

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

all BaseType elements, selecting only those BaseType elements that are not present in the view (this can be accomplished by checking for a null value). Compared to the response time of the first implementation (1517.4 seconds), this rewriting query results in much less response time (23.38 seconds) as reported in Figure 10.

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

The performance of the value-based join queries, QJ1-QJ4, is shown in Figure 11.

Both CNX and Timber show poor performance on these “traditional” join queries. In Timber, a simple, unoptimized nested loop join algorithm is used to evaluate value-based joins. Both QJ1 and QJ2 perform poorly because of the high overhead in retrieving attribute values through random accesses.

CNX has poor overall performance compared to the other two databases on the value based join queries (QJ1-QJ2) for the small scale version of the database, which is due to the fact that the joins are carried out in a naive nested loop join. Thus, the complexity of the queries is $O(n^2)$. The selectivity factor has much impact on the value based joins in CNX. Notice that CNX scales up better than Timber on QJ1 and QJ2, where CNX results in super linear scale up, while Timber scales up very poorly and COR has linear scale-up. For pointer based join queries (QJ3 and QJ4), CNX compares poorly to the COR, although it still shows a super-linear scale-up curve with respect to the size of the database.

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

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

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

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

In COR, 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.

With CNX being a native XML database, on ewould expect it to perform reasonably well on structural aggregation queries, such as QA4. The reason for the poor performance of CNX on QA4 is due to the strategy of the query execution: it is essentially two levels of nested loops join involving three aggregate functions (two

| Query | Query Description | Sel.(%) | Response Times (seconds) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | ds0.1x | | | | ds1x | | | |
| | | | CNX | | Timber | COR | CNX | | Timber | COR |
| | | | Idx | No Idx | | | Idx | No Idx | | |
| QA1 | Value aggregation | 1 node | 15.35 | 12.32 | 12.12 | 0.01 | 170.04 | 154.31 | 1184.69 | 0.11 |
| QA2 | Value aggregation with groupby | 16 nodes | N/A | N/A | 10.11 | 0.06 | N/A | N/A | N/A | 0.54 |
| QA3 | Value aggregate selection | 0.3 | N/A | N/A | N/A | 18.14 | N/A | N/A | N/A | 201.43 |
| QA4 | Structural aggregation | 0.02 | 358.46 | 357.75 | 0.04 | 0.39 | 1298.22 | 1282.51 | N/A | 3.55 |
| QA5 | Structural aggregate selection | 3.1 | 3.03 | 2.70 | 15.38 | 0.26 | 8.19 | 7.45 | 1288.67 | 3.65 |
| QA6 | Structural exploration | 0.4 | N/A | N/A | 0.07 | 12.00 | N/A | N/A | 34.17 | 132.59 |

Figure 12: Benchmark Numbers for Three DBMSs on Aggregate Queries

| Query | Query Description | Response Times (seconds) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | ds0.1x | | | | ds1x | | | |
| | | CNX | | Timber | COR | CNX | | Timber | COR |
| | | Idx | No Idx | | | Idx | No Idx | | |
| QU1 | Point insert | N/A | N/A | N/A | 0.55 | N/A | N/A | N/A | 5.72 |
| QU2 | Point delete | N/A | N/A | N/A | 2.31 | N/A | N/A | N/A | 163.94 |
| QU3 | Bulk insert | N/A | N/A | N/A | 4.67 | N/A | N/A | N/A | 41.84 |
| QU4 | Bulk delete | N/A | N/A | N/A | 0.79 | N/A | N/A | N/A | 43.76 |
| QU5 | Bulk load | N/A | N/A | N/A | 60.00 | N/A | N/A | N/A | 807.33 |
| QU6 | Bulk Construction | N/A | N/A | N/A | 0.83 | N/A | N/A | N/A | 2.48 |
| QU7 | Restructuring | N/A | N/A | N/A | 13.81 | N/A | N/A | N/A | 728.61 |

Figure 13: Benchmark Numbers for Three DBMSs on Update Queries

*count()* functions and one *max()* function), which dramatically increase the response time. For other simpler queries (QA1, QA5), the performance of CNX is better than that of Timber, but worse than than that of COR.

### 5.2.12 Update (QU1-QU7)

Figure 13 shows the performance of update queries, QU1-QU7.

In COR, the query time for the point insert query (QU1) is less than the query time for the point delete query (QU2) because the children of the deleted node is needed to be update while there is no children of the newly inserted node. This is also true for the performance difference between the delete query (QU2) and the bulk delete query (QU4). In the bulk delete query (QU4), we simply delete all leaf nodes with aSixteen = 3. The bulk loading (QU5) takes a long time because each row corresponding to each element needs to be inserted. The time for the query QU6 does not entirely reflect the actual bulk reconstruction since COR does not yet have a function available to group the content of elements together to reconstruct an XML document. The restructuring query (QU7) takes an excessive amount of time because finding and updating the descendants of the given element require nested loop joins and a large number of row scans.

Updates are not supported in CNX and Timber.

## 5.3 Performance Analysis on Scaling Databases

In this Section, we discuss the performance of the three systems as the data set is scaled from **ds0.1x** to **ds1x**. Please refer to Figure 5 for the performance comparison between these two data sets.

### 5.3.1 Scaling Performance on CNX

In almost all of the queries, the ratios of the response times when using ds0.1x over ds1x are less than or around 10, except for QS35, which consists of nested aggregate count() function. This indicates that CNX scales at least linearly, and sometimes super-linearly with respect to the database size growth. The reason for long response time of the join queries QJ1 - QJ4 is that the XPath expressions executed on CNX invoke a nested loop join, which its the complexity is the order of $n^2$.

### 5.3.2 Scaling Performance on Timber

Timber scales linearly for all queries, with a response time ratio of approximately 10, with two exceptions. Where large return result structures have to be constructed, Timber is inefficient, and scales poorly, as discussed above in Section 5.2.1. Also, the value-based join implementation is naive, and scales poorly.

### 5.3.3 Scaling Performance on COR

Once more, with two exceptions, the ratios of the response times when using **ds0.1x** over **ds1x** are approximately 10, showing linear scale-up. Exceptions to this occur in two types of queries: a) the returned structure with descendants queries, b) the text processing queries, and c) the update queries.

QR3 and QR4 require result XML reconstruction with descendant access, and have response times grow about 20 times as data size increases about 10 times. Recently, Shanmugasundaram et al. [17,23] have addressed this problem as they proposed techniques for efficiently publishing and querying XML view of relational data. However, these techniques were not implemented in COR.

Text-processing queries also scale poorly. There are two types of text processing queries: element content selection (QS11-QS12), and string distance selection (QS13-QS14). Queries QS11 and QS12 are single table queries that use a `LIKE` predicate. The attribute being queried does not have an index in both data sets (the index wizard chose not to build an index on this attribute). Consequently, in both cases a full scan of the table is required. The same behavior is seen for queries QS13 and QS14, which also use table scans. However, instead of using a `LIKE` predicate (as QS12 did), they use a user-defined function. The costs of these queries also increases faster than the table size.

Figure 13 indicates that the performance gets worse as data size increases for most of the update queries. This is because of the complexity of finding and updating the elements that are related to the deleted or inserted elements. Most of joins used in these update queries are nested loop joins which grow exponentially respective to the input sizes.

## 6 Conclusions

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

We have used the benchmark to evaluate three XML systems: a commercial XML system, Timber, and a commercial Object-Relational DBMS. The results show that the commercial native XML system has substantial room for performance improvement on most of the queries. The benchmark has already become an invaluable tool in the development of the Timber native XML database, helping us identify portions of the system that need performance tuning. Consequently, on most benchmark queries Timber outperforms the other systems. A notable exception to this behavior is the poor performance of Timber on traditional value-based join queries.

This benchmarking effort also shows that the ORDBMS is sensitive to the method used to translate an XML query to SQL. While this has been shown to be true for some XML queries in the past [14, 17], we show that this is also true for simple indirect containment queries, and queries that search for irregular structures. We also demonstrate that using recursive SQL one can evaluate any structural query in the benchmark, however, this is much more expensive in the ORDBMS than the implementations in Timber, which use efficient XML structural join algorithms.

Finally, we note that the proposed benchmark meets the key criteria for a successful domain-specific benchmark that have been proposed in [16]. 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 [26]. It is *scalable* through the use of a scaling parameter. It is *simple* since it comprises only one data set and a set of simple queries, each designed to test a distinct functionality.

## References

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

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

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

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

[5] B. B. Yao and M. Tamer Özsu and J. Keenleyside. XBench – A Family of Benchmarks for XML DBMSs. In *VLDB EEXTT Workshop*, 2002.

[6] 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.

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

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

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

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

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

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

## A Query Selectivity Computation

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 (1.6%) since they are selected based on aSixtyFour attribute which has a probability of 1/64.

**QS1.** Select nodes with aString = "Sing a song of oneB4". Selectivity is 1/128 (0.8%) since the probability of "oneB4" is 1/128.

**QS2.** Select nodes with aString = "Sing a song of oneB1". Selectivity is 1/16 (6.3%) since the probability of "oneB1" is 1/16.

**QS3.** 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.

**QS4.** 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.

**QS5.** Select nodes with aSixtyFour between 5 and 8. Selectivity is $4 \times 1/64 = 1/16(6.3\%)$.

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

**QS7.** Select nodes with attributes aSixteen = 1 and aFour = 1. Selectivity is $1/16 \times 1/4 = 1/64$ (1.6%) since the selectivity of nodes with aLevel = 13 is 6.0%.

**QS8.** Select nodes based on the element name, eOccasional. Selectivity is $1/64(1.6\%)$ since eOccasional appears nested under the element with aSixtyFour = 0 and the probability of aSixtyFour = 0 is 1/16.

**QS9.** 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%.

**QS10.** Select the second child of every node with aLevel = 9. At level 9, 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%.

**QS11.** 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%).

**QS12.** 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%).

**QS13.** Select all nodes with element content that the distance between keyword "oneB5" and keyword "twenty" is not more than four. The probability of any one occurrence of "oneB5" being selected is 1/256. There are two placeholders that "oneB5" can be at and that has the distance to "twenty" not more than four. Thus, the overall selectivity is $(1/256) \times 2 = 1/128$ (0.8%).

**QS14.** 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 $(1/32) \times 2 = 1/16$ (6.3%).

**QS15.** 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 $(1/2) \times (1/16) = 1/32$ (3.1%).

**QS16.** Select the second element with aFour = 1 below any element with aSixtyFour = 1. This query returns at most one element.

**QS17.** Among the children with aSixteen = 1 of the parent element with aLevel = 13, select the last one. Approximately 5.95% 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 $0.0595 \times 1/8$ = 0.7%.

**QS18.** Select nodes with aLevel = 13 that have a child with aSixteen = 3. The first predicate has a selectivity of 5.95%, and the second predicate 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 $0.0595 \times (1/16) \times 2$ = 0.7%.

**QS19.** Select nodes with aLevel = 15 that have a child with aSixtyFour = 3. The first predicate has a selectivity of 23.78%, and the second predicate has a selectivity of 1/64. Following the same argument as above, the selectivity of the query as a whole is still 0.7%.

**QS20.** Select nodes with aLevel = 11 that have a child with aFour = 3. The first predicate has a selectivity of 1.49%, and the second predicate has a selectivity of 1/4. Following the same argument as above, the selectivity of the query as a whole is still 0.7%.

**QS21.** Select nodes with aLevel = 13 that have a descendant with aSixteen = 3. The first predicate has selectivity of 0.0595. 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 $0.0595 \times (1-(1-(1/16))^{14})$ = 3.5%.

**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 themselves based on the equality of aUnique1 attribute. The probability of aSixtyFour = 2 is 1/64, thus the selectivity of this query is 1/64 (1.6%).

**QJ2.** Select nodes with aSixteen = 2 and join with themselves based on the equality of aLevel attribute. The probability of aSixteen = 2 is 1/16, thus the selectivity of this query is 1/16 (6.3%).

**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.** 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 $P_n$("oneB1") be the probability that there are $n$ occurrences of "oneB1." Then, $P_n$("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 $1 - P_0$("oneB1") - $P_1$("oneB1") = 1 - 0.36 - 0.38 = 0.3. Thus, the selectivity of this query is 0.3%.

**QA4.** 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%).

**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%.

## B SQL Queries for Mbench

First, we show the SQL schema for the SQL queries and then we present each SQL query.

The SQL schema is as follows:

```
CREATE TABLE eNest(eNest_ID integer not null,
                   eNest_parentID  integer not null,
                   eNest_parentCODE varchar(50),
                   eNest_childOrder integer not null,
                   eNest_aUnique1 integer not null,
                   eNest_aUnique2 integer not null,
                   eNest_aLevel integer not null,
                   eNest_aFour integer not null,
                   eNest_aSixteen integer not null,
                   eNest_aSixtyFour integer not null,
                   eNest_aString varchar(40),
                   eNest_val varchar(600),
                   primary key (eNest_ID));

CREATE TABLE eOccasional(eOccasional_ID integer not null,
                   eOccasional_parentID  integer not null,
                   eOccasional_parentCODE varchar(50),
                   eOccasional_childOrder integer not null,
                   eOccasional_aRef integer not null,
                   eOccasional_val varchar(550),
                   primary key (eOccasional_ID));
```

- QR1: Select all elements with aSixtyFour = 2 (Return only the element in question

```
select eNest_aUnique1
from eNest
where eNest_aSixtyFour = 2;
```

- QR2: Select all elements with aSixtyFour = 2 (Return the element and all its immediate children)

```
create table tmp2_qr2(aUnique1ID integer, ID integer);
create table tmp1_qr2(parentID integer, childID integer);

-- contains all elements with aSixtyFour = 2
delete from tmp2_qr2;
insert into tmp2_qr2
select eNest_aUnique1, eNest_ID
from eNest
where eNest_aSixtyFour = 2;

-- contains elements with aSixtyFour = 2 that have eNest children
delete from tmp1_qr2;
insert into tmp1_qr2
select p.aUnique1ID as parentID, c.eNest_aUnique1 as childID
from tmp2_qr2 p
left outer join
     eNest c
on p.ID = c.eNest_parentID;

insert into tmp1_qr2
select p.aUnique1ID as parentID, c.eOccasional_aRef as childID
from tmp2_qr2 p,
     eOccasional c
where p.ID = c.eOccasional_parentID;

select parentID, count(childID)
from tmp1_qr2
group by parentID;
```

- QR3: Select all elements with aSixtyFour = 2 (Return the entire subtree)

```
drop view tmp1_qr3;
create view tmp1_qr3 as
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’ancestor’  to
-- store the descendants of ’eNest’ nodes with
-- ’aSixtyFour’ = 2
select rootID, ID
from ancestor;

select r.eNest_aUnique1, count(d.eNest_aUnique1)
```

```
from eNest r, eNest d, tmp1_qr3
where r.eNest_ID = tmp1_qr3.rootID
and d.eNest_ID = tmp1_qr3.ID
group by r.eNest_aUnique1;
```

- QR4: Select all elements with aSixtyFour = 2 and selected descendants with aFour = 1

```
create table tmp2_qr4(aUnique1 integer, numD integer);
create table tmp3_qr4(aUnique1 integer, numD integer);

drop view tmp1_qr4;
create view tmp1_qr4 as
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’ancestor’  to
-- store the descendants with ’aFour’ = 1 with
-- ’eNest’ nodes with ’aSixtyFour’ = 2
select rootID, ID
from ancestor;

delete from tmp2_qr4;
insert into tmp2_qr4
select r.eNest_aUnique1, count(d.eNest_aUnique1)
from eNest r, eNest d, tmp1_qr4
where r.eNest_ID = tmp1_qr4.rootID
and d.eNest_ID = tmp1_qr4.ID
and d.eNest_aFour = 1
group by r.eNest_aUnique1;

delete from tmp3_qr4;
insert into tmp3_qr4
select eNest_aUnique1,0
from eNest
where eNest_aSixtyFour = 2;

select t1.aUnique1, t2.numD
from tmp3_qr4 t1
left outer join
tmp2_qr4 t2
on t1.aUnique1 = t2.aUnique1;
```

- QS1: Select elements with aString = ’Sing a song of oneB4’

```
select eNest_aUnique1
from eNest e
where eNest_aString = ’Sing a song of oneB4’;
```

- QS2: Select elements with aString = ’Sing a song of oneB1’

```
select eNest_aUnique1
from eNest e
where eNest_aString = ’Sing a song of oneB1’;
```

- QS3: Select elements with aLevel = 10

```
select eNest_aUnique1
from eNest e
where eNest_aLevel = 10;
```

- QS4: Select elements with aLevel = 13

```
select eNest_aUnique1
from eNest e
where eNest_aLevel = 13;
```

- QS5: Select nodes that have aSixtyFour between 5 and 8.

```
select eNest_aUnique1
from eNest
where eNest_aSixtyFour between 5 and 8;
```

- QS6: Select nodes with aLevel = a13 and have the returned nodes sorted by aSixtyFour attribute.

```
select eNest_aUnique1
from eNest
where eNest_aLevel = 13
order by eNest_aSixtyFour;
```

- QS7: Select nodes with aSixteen = 1 and aFour = 1.

```
select eNest_aUnique1
from eNest e
where eNest_aFour = 1
and eNest_aSixteen = 1;
```

- QS8: Select nodes with the element name, eOccasional

```
select eOccasional_aRef
from eOccasional e;
```

- QS9: Select the second child of every node with aLevel = 7

```
select eChild.eNest_aUnique1
from eNest eParent, eNest eChild
where eParent.eNest_aLevel = 7
and eChild.eNest_childOrder = 2
and eParent.eNest_ID = eChild.eNest_parentID
and eParent.eNest_parentCODE = ’eNest’;
```

- QS10: Select the second child of every node with aLevel = 9

```
select eChild.eNest_aUnique1
from eNest eParent, eNest eChild
where eParent.eNest_aLevel = 9
and eChild.eNest_childOrder = 2
and eParent.eNest_ID = eChild.eNest_parentID
and eParent.eNest_parentCODE = ’eNest’;
```

- QS11: Get ‘eOccasional’ nodes that have element content contains “oneB4”

```
select eOccasional_aRef
from eOccasional e
where eOccasional_val like ’%oneB4%’;
```

- QS12: Get nodes that have element content contains “oneB4”

  ```
  select eNest_aUnique1
  from eNest e
  where eNest_val like ’%oneB4%’;
  ```

- QS13: select all nodes with element content that the distance between keyword “oneB5” and the keyword “twenty” is not more than four

  ```
  select eNest_aUnique1
  from eNest
  where eNest_val like ’%oneB5%’
  and isRightDist(eNest_val, ’twenty ’, ’oneB5’,4) = 1;
  ```

- QS14: select all nodes with element content that the distance between keyword “oneB2” and the keyword “twenty” is not more than four

  ```
  select eNest_aUnique1
  from eNest
  where eNest_val like ’%oneB2%’
  and isRightDist(eNest_val, ’twenty ’, ’oneB2’,4) = 1;
  ```

- QS15: Select the second element below each element with aFour = 1 if that second element also has aFour = 1.

  ```
  select eParent.eNest_aUnique1
  from eNest eParent, eNest eChild
  where eParent.eNest_aFour =  1
  and eChild.eNest_aFour = 1
  and eParent.eNest_ID = eChild.eNest_parentID
  and eParent.eNest_parentCODE = ’eNest’
  and eChild.eNest_childOrder = 2;
  ```

- QS16: Select the second element with aFour = 1 below any element with aSixtyFour = 1

  ```
  -- 1) Select element with aSixtyFour = 1 -- 2) Select the second
  element of the element in 1) --    and fetch only  the first row

  select eChild.eNest_aUnique1
  from eNest eChild,
  (select eParent.eNest_ID
  from eNest eParent
  where eParent.eNest_aSixtyFour = 1) as tmp(eParentID)
  where eChild.eNest_aFour = 1
  and eChild.eNest_childOrder = 2
  and eChild.eNest_parentID = eParentID
  and eChild.eNest_parentCODE = ’eNest’ fetch first 1 rows only;
  ```

- QS17: Reverse ordering. Among the children with aSixteen = 1 of the parent element with aLevel = 13, select the last child.

  ```
  -- 1) find the ID of element that has the last order -- 2) get the
  element that has the matching ID --   found in 1) and that has
  aSixteen = 1

  drop view tmp1_qs20;
  ```

```
create view tmp1_qs20(parentID, ID, cOrder)
as select eChild.eNest_parentID, eChild.eNest_ID,
       eChild.eNest_childOrder
from eNest eParent, eNest eChild
where eParent.eNest_aLevel = 13
and eChild.eNest_aSixteen = 1
and eChild.eNest_parentID = eParent.eNest_ID
and eChild.eNest_parentCODE = 'eNest';


drop view tmp3_qs20;
create view tmp3_qs20(parentID, ID) as
select t1.parentID, ID
from tmp1_qs20 t1,
(select parentID, max(cOrder)
from tmp1_qs20 t2
group by t2.parentID) as tmp2_qs20(parentID, order)
where t1.parentID = tmp2_qs20.parentID
and t1.cOrder = tmp2_qs20.order;

select eNest_aUnique1
from eNest c, tmp3_qs20
where c.eNest_ID = tmp3_qs20.ID;
```

- QS18: Select nodes with aLevel = 13 that have a child with aSixteen = 3

```
select eParent.eNest_aUnique1
from eNest eParent, eNest eChild
where eParent.eNest_aLevel =  13
and eChild.eNest_aSixteen = 3
and eParent.eNest_ID = eChild.eNest_parentID
and eParent.eNest_parentCODE = 'eNest';
```

- QS19: Select nodes with aLevel = 15 that have a child with aSixtyFour = 3

```
select eParent.eNest_aUnique1
from eNest eParent, eNest eChild
where eParent.eNest_aLevel =  15
and eChild.eNest_aSixtyFour = 3
and eParent.eNest_ID = eChild.eNest_parentID
and eParent.eNest_parentCODE = 'eNest';
```

- QS20: Select nodes with aLevel = 11 that have a child with aFour = 3

```
select eParent.eNest_aUnique1
from eNest eParent, eNest eChild
where eParent.eNest_aLevel =  11
and eChild.eNest_aFour = 3
and eParent.eNest_ID = eChild.eNest_parentID
and eParent.eNest_parentCODE = 'eNest';
```

- QS21: Select nodes with aLevel = 13 that have a descendant with aSixteen = 3

```
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’ancestor’ to
-- store the ancestor of ’eNest’ nodes
-- with ’aSixteen’ = 3
select distinct eNest_aUnique1
from eNest, ancestor
where eNest.eNest_ID = ancestor.ID
and eNest.eNest_aLevel = 13;
```

- QS22: Select nodes with aLevel = 15 that have a descendant with aSixtyFour = 3

```
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’ancestor’ to
-- store the ancestor of ’eNest’ nodes
-- with ’aSixtyFour’ = 3
select distinct eNest_aUnique1
from eNest, ancestor
where eNest.eNest_ID = ancestor.ID
and eNest.eNest_aLevel = 15;
```

- QS23: Select nodes with aLevel = 11 that have a descendant with aFour = 3

```
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’descendant’ to
-- store the descendants of ’eNest’ nodes
-- with ’aLevel’ = 11
select distinct e1.eNest_aUnique1
from descendant d, eNest e1, eNest e2
where d.rootID = e1.eNest_ID
and e1.eNest_aLevel = 11
and d.ID = e2.eNest_ID
and e2.eNest_aFour = 3;
```

- QS24: Select nodes with aSixteen = 3 that have a descendant with aSixteen = 5

```
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’ancestor’ to
-- store the ancestor of ’eNest’ nodes with ’aSixteen’ = 5
select distinct eNest_aUnique1
from eNest, ancestor
where eNest.eNest_ID = ancestor.ID
and eNest.eNest_aSixteen = 3;
```

- QS25: Select nodes with aFour = 3 that have a descendant with aSixtyFour= 3

```
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’tmp1_qs25’ to
-- store the ancestor of ’eNest’ nodes
-- with ’aSixtyFour’ = 3
select distinct eNest_aUnique1
from eNest, ancestor
where eNest.eNest_ID = ancestor.ID
and eNest.eNest_aFour = 3;
```

- QS26: Select nodes with aSixtyFour = 9 that have a descendant with aFour = 3

```
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’descendant’ to
-- store the descendants of ’eNest’ nodes
-- with ’aFour’ = 3
select distinct eNest_aUnique1
from eNest e1, eNest e2, descendant d
where d.rootID = e1.eNest_ID
and e1.eNest_aSixtyFour = 9
and d.ID = e2.eNest_ID
and eNest.eNest_aFour = 3;
```

- QS27: Select nodes with aSixtyFour = 9 that have a descendant with aFour = 3. Return a pair of ancestor and descendant nodes

```
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’descendant’ to
-- store the ancestor of ’eNest’ nodes
-- with ’aFour’ = 3
select distinct e1.eNest_aUnique1, e2.eNest_aUnique1
from eNest e1, eNest e2, descendant d
where d.rootID = e1.eNest_ID
and e1.eNest_aSixtyFour = 9
and d.ID = e2.eNest_ID
and eNest.eNest_aFour = 3;
```

- QS28: 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

```
select distinct node1.eNest_aUnique1
from eNest node1, eNest node2, eNest node3,
     eNest node4
where node1.eNest_aFour = 3
and node2.eNest_aSixteen = 3
and node3.eNest_aSixteen = 5
and node4.eNest_aLevel = 16
and node2.eNest_parentID = node1.eNest_ID
and node2.eNest_parentCODE = ’eNest’
and node3.eNest_parentID = node2.eNest_ID
and node3.eNest_parentCODE = ’eNest’
and node4.eNest_parentID = node3.eNest_ID
and node4.eNest_parentCODE = ’eNest’;
```

- QS29: 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

```
select distinct eParent.eNest_aUnique1
from eNest eParent, eNest eChild1, eNest eChild2
where eParent.eNest_aLevel = 11
and eChild1.eNest_aFour = 3
and eChild2.eNest_aSixtyFour = 3
and eChild1.eNest_parentID = eParent.eNest_ID
and eChild1.eNest_parentCODE = ’eNest’
and eChild2.eNest_parentID = eParent.eNest_ID
and eChild2.eNest_parentCODE = ’eNest’
and eChild1.eNest_ID <> eChild2.eNest_ID;
```

- QS30: One twig query with two parent child selection, low selectivity of parent aFour = 1, low selectivity of left child aLevel = 11, and low selectivity of right child aSixtyFour = 3

```
select distinct eParent.eNest_aUnique1
from eNest eParent, eNest eChild1, eNest eChild2
where eParent.eNest_aFour = 1
and eChild1.eNest_aLevel = 11
and eChild2.eNest_aSixtyFour = 3
and eChild1.eNest_parentID = eParent.eNest_ID
and eChild1.eNest_parentCODE = ’eNest’
and eChild2.eNest_parentID = eParent.eNest_ID
and eChild2.eNest_parentCODE = ’eNest’
and eChild1.eNest_ID <> eChild2.eNest_ID;
```

- QS31: One chain query with three ancestor-descendant 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

```
create table tmp2_qs31(aID integer);
create table tmp3_qs31(aID integer);
create table tmp4_qs31(aID integer);
create table tmp5_qs31(ID integer);

drop view tmp1_qs31
create view tmp1_qs31(aID) as
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’ancestor1’ to
-- store the ancestors of ’eNest’ nodes
-- with ’aLevel’ = 16
select distinct ID
from ancestor1;


delete from tmp2_qs31;
insert into tmp2_qs31
select a.eNest_ID as aID
from eNest a, tmp1_qs31
where a.eNest_ID = tmp1_qs31.ID
and a.eNest_aSixteen = 5;


delete from tmp3_qs31;
insert into tmp3_qs31
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’tmp2_qs31’ to
-- store the ancestors of ’eNest’ nodes
-- with ’aLevel’ = 16 that have ancestor nodes
-- with ’aSixteen’ = 5 in ’tmp3_qs31’
```

```
select distinct ID
from ancestor2;

delete from tmp4_qs31;
insert into tmp4_qs31
select a.eNest_ID as aID
from eNest a, tmp3_qs31
where a.eNest_ID = tmp3_qs31.ID
and a.eNest_aSixteen = 3;

-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’tmp4_qs31’ to
-- store the ancestors of ’eNest’ nodes with
-- ’aLevel’ = 16 that have ancestor nodes with
-- ’aSixteen’ = 5, and that have ancestor nodes with
-- ’aSixteen’ = 3 in ’tmp5_qs31’
select distinct a.eNest_aUnique1
from eNest a, tmp5_qs31
where a.eNest_ID = tmp5_qs31.ID
and a.eNest_aFour =  3;
```

- QS32: One twig query with two ancestor descendant selection, low selectivity of ancestor aLevel = 11, high selectivity of one descendant aFour = 3, and low selectivity of another descendant aSixtyFour = 3

```
create table tmp1_qs32(aUnique1 integer);
create table tmp2_qs32(aUnique1 integer);

delete from tmp1_qs32;
insert into tmp1_qs32
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’tmp1_qs32’ to
-- store the ancestors of ’eNest’ nodes
-- with ’aSixtyFour’ = 3
select distinct eNest_aUnique1
from ancestor, eNest
where eNest_aLevel = 11
and eNest_ID = ID;

delete from tmp2_qs32;
insert into tmp2_qs32
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’tmp2_qs32’ to
-- store the ancestors of ’eNest’ nodes
-- with ’aFour’ = 3
select distinct eNest_aUnique1
from ancestor, eNest
where eNest_aLevel = 11
and eNest_ID = ID;

select tmp1_qs32.aUnique1
from tmp1_qs32, tmp2_qs32
```

```
where tmp1_qs32.aUnique1 = tmp2_qs32.aUnique1;
```

- QS33: One twig query with two ancestor descendant selection, low selectivity of ancestor aFour = 1, low selectivity of one descendant aLevel = 11, and low selectivity of another descendant aSixtyFour = 3

```
drop view tmp1_qs33;
create view tmp1_qs33(ID, aUnique1) as
-- vendor specific call for recursive query removed
-- recursive join 'eNest' table and 'ancestor' to
-- store the ancestors of 'eNest' nodes
-- with 'aLevel' = 11
select distinct ID, eNest_aUnique1
from ancestor, eNest
where eNest_aFour = 1
and eNest_ID = ID;

drop view tmp2_qs33;
create view tmp2_qs33(ID, aUnique1) as
-- vendor specific call for recursive query removed
-- recursive join 'eNest' table and 'ancestor' to
-- store the ancestors of 'eNest' nodes
-- with 'aSixtyFour' = 3
select distinct ID, eNest_aUnique1
from ancestor, eNest
where eNest_aFour = 1
and eNest_ID = ID;

select tmp1_qs33.aUnique1
from tmp1_qs33, tmp2_qs33
where tmp1_qs33.ID = tmp2_qs33.ID;
```

- QS34: One twig query with two ancestor descendant selection, low selectivity of ancestor aFour = 1, low selectivity of a child with aLevel = 11, and low selectivity of another descendant aSixtyFour = 3

```
-- vendor specific call for recursive query removed
-- recursive join 'eNest' table and 'ancestor' to
-- store the ancestors of 'eNest' nodes
-- with 'aSixtyFour' = 3
select distinct a.eNest_aUnique1
from eNest a, ancestor, eNest c
where a.eNest_ID = ancestor.ID
and c.eNest_parentID = a.eNest_ID
and c.eNest_parentCODE = 'eNest'
and c.eNest_aLevel = 11
and a.eNest_aFour = 1;
```

- QS35: Missing Elements. Find all BaseType elements such that there is no OccasionalType elements below them.

  1) Find all BaseType elements that have some OccasionalType elements below them.

  2) Left outer join between all BaseType elements and BaseType elements that have OccasionalType elements below them.

```
create table tmp1(ID integer, hasOccasional integer);

delete from tmp1;
insert into tmp1;
-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’subeNest’ to
-- store ’eNest’ nodes that have ’eOccasional’
-- as descendants
select distinct ID, hasOccasional
from subeNest;

drop view tmp2;
create view tmp2(aUnique1, hasOccasional) as
select eNest_aUnique1, t.hasOccasional
from eNest c
left outer join
tmp t
on t.ID = c.eNest_ID;

select aUnique1 from tmp2
where hasOccasional is null;
```

- QJ1: Select nodes based on aSixtyFour = 2 and join with themselves based on the equality value of aUnique1.

```
select e1.eNest_aUnique1, e2.eNest_aUnique1
from eNest e1, eNest e2
where e1.eNest_aSixtyFour = 2
and e1.eNest_aSixtyFour = e2.eNest_aSixtyFour
and e1.eNest_aUnique1 = e2.eNest_aUnique1;
```

- QJ2: Select nodes based on aSixteen = 2 and join with themselves based on the equality value of aUnique1.

```
select e1.eNest_aUnique1, e2.eNest_aUnique1
from eNest e1, eNest e2
where e1.eNest_aSixteen = 2
and e1.eNest_aSixteen = e2.eNest_aSixteen
and e1.eNest_aUnique1 = e2.eNest_aUnique1;
```

- QJ3: Select all OccasionalType nodes that point to a node with aSixtyFour = 3

```
select eOccasional_aRef
from eOccasional, eNest
where eNest_aSixtyFour = 3
and eNest_aUnique1 = eOccasional_aRef;
```

- QJ4: Select all OccasionalType nodes that point to a node with aFour = 3

```
select eOccasional_aRef
from eOccasional, eNest
where eNest_aFour = 3
and eNest_aUnique1 = eOccasional_aRef;
```

- QA1: Over all nodes at level 15, compute the average value for the aSixtyFour attribute

```
select avg(eNest_aSixtyFour)
from eNest
where eNest_aLevel = 15;
```

- QA2: Over all nodes at all levels, compute the average value for the aSixtyFour attribute

```
select eNest_aLevel, avg(eNest_aSixtyFour)
from eNest
group by eNest_aLevel;
```

- QA3: Select elements that have at least two occurrences of keyword “oneB1” in their content

```
select eNest_aUnique1
from eNest
where eNest_val like ’%oneB1%’
and isNumKeysGTE(eNest_val,’oneB1’,2) = 1;
```

- QA4: Amongst the nodes at level 11, find the node(s) with the largest fanout.

  1) find number of children of each node at level 11

  2) find nodes that have the number of children equal to the the largest number of children.

```
CREATE TABLE tmp1_qa3(pID integer, cID integer);
CREATE TABLE tmp2_qa3(pID integer, numC integer);

delete from tmp1_qa3;
insert into tmp1_qa3;
select distinct p.eNest_ID, c.eNest_ID
from eNest p, eNest c
where c.eNest_parentID = p.eNest_ID
and p.eNest_aLevel = 11
union
select distinct p.eNest_ID, c.eOccasional_aRef
from eNest p, eOccasional c
where c.eOccasional_parentID = p.eNest_ID
and p.eNest_aLevel = 11;

delete from tmp2_qa3;
insert into tmp2_qa3
select pID, count(cID)
from tmp1_qa3
group by pID;

drop view tmp3_qa3;
create view tmp3_qa3(maxNumC) as
select distinct max(numC)
from tmp2_qa3;

select distinct eNest_aUnique1
from eNest, tmp2_qa3, tmp3_qa3
where eNest.eNest_ID = tmp2_qa3.pID
and numC = maxNumC;
```

- QA5: select elements that have at least two children that satisfy aFour = 1

```
select distinct eParent.eNest_aUnique1
from eNest eParent, eNest eChild1, eNest eChild2
where eChild1.eNest_aFour = 1
and eChild2.eNest_aFour = 1
and eChild1.eNest_parentID = eParent.eNest_ID
and eChild1.eNest_parentCODE = ’eNest’
and eChild2.eNest_parentID = eParent.eNest_ID
and eChild2.eNest_parentCODE = ’eNest’
and eChild1.eNest_ID <> eChild2.eNest_ID;
```

- QA6: For each node at level 7, determine the height of the sub-tree rooted at this node

```
drop view tmp1_qa6;
create view tmp1_qa6 as
select max(eNest_aLevel) as maxLevel
from eNest;

-- vendor specific call for recursive query removed
-- recursive join ’eNest’ table and ’tmp1_qa6’  to
-- store the ancestors of ’eNest’ nodes that
-- have maximum level and keep track the height of
-- each node.  At the end, output the ID and the height
-- of nodes with ’aLevel’ = 7
select distinct uniqueID, height
from desc
where level = 7;
```

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

```
drop view tmp1_qu1;
create view tmp1_qu1(parentID, stringVal) as
select eNest_ID, eNest_val
from eNest
where eNest_aUnique1 = 10102;

insert into eNest
select 70000, parentID, ’eNest’, 1, 70000, 3000, 10, 3, 15, 60,
’Sing a song of oneB11’, stringVal
from tmp1_qu1;
```

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

```
update eNest
set eNest_parentID =
(select eNest_parentID
from eNest
where eNest_aUnique1 = 10102)
where eNest_parentID =
(select eNest_ID
from eNest
where eNest_aUnique1 = 10102);

-- delete the node with aUnique1 = 10102
```

```
delete from eNest
where eNest_aUnique1 = 10102;
```

- QU3: 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.

```
drop view tmp1_qu3;
create view tmp1_qu3(eNest_ID, eNest_parentID, eNest_parentCODE,
                     eNest_childOrder, eNest_aUnique1, eNest_aUnique2,
                     eNest_aLevel, eNest_aFour, eNest_aSixteen,
                     eNest_aSixtyFour, eNest_aString, eNest_val) as
select eNest_ID + 70000, eNest_ID, eNest_parentCODE, eNest_childOrder,
        eNest_aUnique1, eNest_aUnique2, eNest_aLevel, eNest_aFour,
        eNest_aSixteen, eNest_aSixtyFour, eNest_aString, eNest_val
from eNest
where eNest_aSixtyFour = 1;

insert into eNest
select *
from tmp1_qu3;
```

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

```
delete from eNest
where eNest_parentID not in
(select eNest_ID
from eNest)
and eNest_aSixteen = 3;
```

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

```
load from eNest.txt of del modified by coldel| insert into eNest;
load from eOccasional.txt of del modified by coldel| insert into eOccasional;
```

- **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 OccasionalType.

```
with ancestor(rootID, ID) as
(select eNest_ID, eNest_ID
from eNest, eOccasional
where eNest_ID = eOccasional_parentID
and eNest_aLevel = 11
UNION ALL
select a.rootID, eNest_ID
from eNest e, ancestor a
where a.ID = e.eNest_parentID)
select e.eNest_aUnique1, count(a.ID)
from ancestor a, eNest e
where a.rootID = e.eNest_ID
group by e.eNest_aUnique1;
```

- **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 $u$.

```
drop table tmp1_qu7;

create table tmp1_qu7(eNest_ID integer, eNest_parentCODE varchar(50),
eNest_childOrder integer,
eNest_aUnique1 integer, eNest_aUnique2 integer, eNest_aLevel integer,
eNest_aFour integer,
eNest_aSixteen integer, eNest_aSixtyFour integer,
eNest_aString varchar(40), eNest_val varchar(550));

insert into tmp1_qu7
select eNest_ID, eNest_parentCODE, eNest_childOrder, eNest_aUnique1,
eNest_aUnique2, eNest_aLevel, eNest_aFour, eNest_aSixteen,
eNest_aSixtyFour, eNest_aString, eNest_val
from eNest, eOccasional
where eNest_ID = eOccasional_parentID;

drop table tmp2_qu7;

create table tmp2_qu7(ID integer, parentID integer);

insert into tmp2_qu7
select eNest_ID as ID, eOccasional_ID as parentID
from eOccasional, eNest
where eNest_ID = eOccasional_parentID;

drop table tmp3_qu7;

create table tmp3_qu7(eNest_ID integer, eNest_parentID integer,
eNest_parentCODE varchar(50),
eNest_childOrder integer,
eNest_aUnique1 integer, eNest_aUnique2 integer, eNest_aLevel integer,
eNest_aFour integer,
eNest_aSixteen integer, eNest_aSixtyFour integer,
eNest_aString varchar(40), eNest_val varchar(550));

insert into tmp3_qu7
select t1.eNest_ID, t2.parentID, t1.eNest_parentCODE, t1.eNest_childOrder,
t1.eNest_aUnique1,
t1.eNest_aUnique2, t1.eNest_aLevel, t1.eNest_aFour, t1.eNest_aSixteen,
t1.eNest_aSixtyFour, t1.eNest_aString, t1.eNest_val
from tmp1_qu7 t1, tmp2_qu7 t2
where t1.eNest_ID = t2.ID;

delete from eNest
where eNest_ID in
(select eOccasional_parentID
from eOccasional);

insert into eNest
select * from tmp3_qu7;
```

```
update eOccasional u
set u.eOccasional_parentID =
(select v.eNest_parentID
from eNest v
where v.eNest_ID = u.eOccasional_parentID);
```

## C XPath Queries for MBench

- QR1: Select all elements with aSixtyFour = 2 (Return only the element in question)

  ```
  //eNest[@aSixtyFour=2]/@aUnique1
  ```

- QR2: Select all elements with aSixtyFour = 2 (Return the element and all its immediate children)

  ```
  //eNest[@aSixtyFour=2]
  ```

- QR3: Select all elements with aSixtyFour = 2 (Return the entire subtree)

  ```
  //eNest[@aSixtyFour=2]/@aUnique1
  ```

- QR4: Select all elements with aSixtyFour = 2 and selected descendants with aFour = 1

  ```
  //eNest[@aSixtyFour=2]/@aUnique1|//eNest[@aSixtyFour=2]//eNest[@aFour=1]/@aUnique
  ```

- QS1: Select elements with aString = ’Sing a song of oneB4’

  ```
  //eNest[@aString = ’Sing a song of oneB4’]/@aUnique1
  ```

- QS2: Select elements with aString = ’Sing a song of oneB1’

  ```
  //eNest[@aString = ’Sing a song of oneB1’]/@aUnique1
  ```

- QS3: Select elements with aLevel = 10

  ```
  //eNest[@aLevel=10]/@aUnique1
  ```

- QS4: Select elements with aLevel = 13

  ```
  //eNest[@aLevel=13]/@aUnique1
  ```

- QS5: Select nodes that have aSixtyFour between 5 and 8

  ```
  //eNest[@aSixtyFour>=5][@aSixtyFour<=8]/@aSixtyFour
  ```

- QS6: Select nodes with aLevel = 13 and have the returned nodes sorted by aSixtyFour attribute

  ```
  //eNest[@aLevel=13] sortby(./@aSixtyFour)/@aSixtyFour
  ```

- QS7: Select nodes with aSixteen = 1 and aFour = 1

  ```
  //eNest[@aSixteen=1][@aFour=1]/@aUnique1
  ```

- QS8: Selection based on the element name, eOccasional

  ```
  //eNest//eOccasional/@aRef
  ```

- QS9: Select the second child of every node with aLevel = 7

  ```
  //eNest[@aLevel=7]/eNest[position()=2]/@aUnique1
  ```

- QS10: Select the second child of every node with aLevel = 9

```
//eNest[@aLevel=9]/eNest[position()=2]/@aUnique1
```

- QS11. Low selectivity. Select OccasionalType nodes that have ”oneB4” in the element content

  ```
  //eOccasional[text()˜="oneB4"]/@aRef
  ```

- QS12. High selectivity. Select nodes that have ”oneB4” as a substring of element content

  ```
  //eNest[text()˜="oneB4"]/@aUnique1
  ```

- QS13. Low selectivity. Select all nodes with element content that the distance between keyword ”oneB5” and keyword ”twenty” is not more than four

  N/A

- QS14. High selectivity. select all nodes with element content that the distance between keyword ”oneB2” and keyword ”twenty” is not more than four

  N/A

- QS15. Local ordering. Select the second element below each element with aFour = 1 (sel= 1/4) if that second element also has aFour = 1

  ```
  //eNest[@aFour=1]/eNest[@aFour=1][position()=2]/@aUnique1
  ```

- QS16. Global ordering. Select the second element with aFour = 1 below any element with aSixtyFour = 1

  ```
  //eNest[@aSixtyFour=1]/eNest[position()=2][@aFour=1]/@aUnique1
  ```

- QS17. Reverse ordering. Among the children with aSixteen = 1 of the parent element with aLevel = 13, select the last child

  ```
  //eNest[@aLevel=13]/eNest[@aSixteen=1][position()=last()]/@aUnique1
  ```

- QS18. Select nodes with aLevel = 13 that have a child with aSixteen = 3

  ```
  //eNest[@aLevel=13][./eNest[@aSixteen=3]]/@aUnique1
  ```

- QS19. Select nodes with aLevel = 15 that have a child with aSixtyFour = 3

  ```
  //eNest[@aLevel=15][./eNest[@aSixteen=3]]/@aUnique1
  ```

- QS20. Select nodes with aLevel = 11 that have a child with aFour = 3

  ```
  //eNest[@aLevel=11][./eNest[@aFour=3]]/@aUnique1
  ```

- QS21. Select nodes with aLevel = 13 that have a descendant with aSixteen = 3

  ```
  //eNest[@aLevel=13][.//eNest[@aSixteen=3]]/@aUnique1
  ```

- QS22. Select nodes with aLevel = 15 that have a descendant with aSixtyFour = 3

  ```
  //eNest[@aLevel=15][.//eNest[@aSixtyFour=3]]/@aUnique1
  ```

- QS23. Select nodes with aLevel = 11 that have a descendant with aFour = 3

  ```
  //eNest[@aLevel=11][.//eNest[@aFour=3]]/@aUnique1
  ```

- QS24. Select nodes with aSixteen = 3 that have a descendant with aSixteen = 5

  ```
  //eNest[@aSixteen=3][.//eNest[@aSixteen=5]]/@aUnique1
  ```

- QS25. Select nodes with aFour = 3 that have a descendant with aSixtyFour = 3

  ```
  //eNest[@aFour=3][.//eNest[@aSixtyFour=3]]/@aUnique1
  ```

- QS26. Select nodes with aSixtyFour = 9 that have a descendant with aFour = 3

  ```
  //eNest[@aSixtyFour=9][.//eNest[@aFour=3]]/@aUnique1
  ```

- QS27. Select nodes with aSixtyFour = 9 that have a descendant with aFour = 3. Return a pair of ancestor and descendant nodes

  ```
  //eNest[@aSixtyFour=9][.//eNest[@aFour=3]]/@aUnique1|//eNest[@aSixtyFour=9]//eNes
  ```

- 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

  ```
  //eNest[@aFour=3][./eNest[@aSixteen=3]/eNest[@aSixteen=5]/eNest[@aLevel=16]]/@aUn
  ```

- QS29. One twig query with two parent-child joins with the selectivity pattern: low-high, low-low. Select parent nodes with aLevel = 11 that have a child with aFour = 3, and another child with aSixtyFour = 3

  ```
  //eNest[@aLevel=11][./eNest[@aFour=3]][./eNest[@aSixtyFour=3]]/@aUnique1
  ```

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

  ```
  //eNest[@aFour=1][./eNest[@aLevel=11]][./eNest[@aSixtyFour=3]]/@aUnique1
  ```

- QS31. One chain query with three ancester descendant 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

  ```
  //eNest[@aFour=3][.//eNest[@aSixteen=3]//eNest[@aSixteen=5]//eNest[@aLevel=16]]/@
  ```

- QS32. One twig query with two ancester-descendant joins with the selectivity pattern: low-high, low-low. Select parent nodes with aLevel = 11 that have a child with aFour = 3, and another child with aSixtyFour = 3

  ```
  //eNest[@aLevel=11][.//eNest[@aFour=3]][.//eNest[@aSixtyFour=3]]/@aUnique1
  ```

- QS33. One twig query with two parent-child joins with the selectivity pattern: high-low, high-low. Select parent nodes with aFour = 1 that have a child with aLevel = 11 and another child with aSixtyFour = 3

  ```
  //eNest[@aFour=1][.//eNest[@aLevel=11]][.//eNest[@aSixtyFour=3]]/@aUnique1
  ```

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

  ```
  //eNest[@aFour=1][./eNest[@aLevel=11]][.//eNest[@aSixtyFour=3]]/@aUnique1
  ```

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

  ```
  //eNest[count(./eOccasional)=0]/@aUnique1
  ```

- QJ1. Select nodes with aSixtyFour = 2 and join with themselves based on the equality of aUnique1 attribute

  ```
  //eNest[@aSixtyFour=2][@aUnique1=//eNest[@aSixtyFour=2]/@aUnique1]/@aUnique1
  ```

- QJ2. Select nodes based on aSixteen = 2 and join with themselves based on the equality of aUnique1 attribute

  ```
  //eNest[@aSixteen=2][@aUnique1=//eNest[@aSixteen=2]/@aUnique1]/@aUnique1
  ```

- QJ3. Select all OccasionalType nodes that point to a node with aSixtyFour = 3

```
//eOccasional[@aRef=//eNest[@aSixtyFour=3]/@aUnique1]/@aRef
```

- QJ4. Select all OccasionalType nodes that point to a node with aFour = 3

  ```
  //eOccasional[@aRef=//eNest[@aFour=3]/@aUnique1]/@aRef
  ```

- QA1. Compute the average value for the aSixtyFour attribute of all nodes at level 15

  ```
  avg(//eNest[@aLevel=15]/@aSixtyFour)
  ```

- QA2. Compute the average value of the aSixtyFour attribute of all nodes at each level.

  ```
  avg(//eNest groupby(@aLevel)/@aSixtyFour)
  ```

- QA3. Value aggregate selection. Select elements that have at least two occurrences of keyword ”oneB1” in their content

  N/A

- QA4. Structural aggregation. Amongst the nodes at level 11 (have aLevel = 11), find the node(s) with the largest fanout

  ```
  //eNest[@aLevel=11][count(./eNest)=max(count(//eNest[@aLevel=11]/eNest))]/@aUniq
  ```

- QA5. Select elements that have at least two children that satisfy aFour = 1

  ```
  //eNest[count(./eNest[@aFour=1])>=2]/@aUnique1
  ```

- QA6. For each node at level 7, determine the height of the sub-tree rooted at this node.

  ```
  N/A
  ```

- QU1-QU7

  N/A