The Michigan Benchmark: Towards XML Query Performance Diagnostics

Kanda Runapongsa   Jignesh M. Patel   H. V. Jagadish

Yun Chen   Shurug Al-Khalifa

University of Michigan

1301 Beal Avenue; 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 DBMSs, 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. In addition, the benchmark also played an important role in helping the development team of Timber (our native XML DBMS) devise a more effective access method, and fine tune the implementation of the structural join algorithms.

1   Introduction

XML query processing has taken on considerable importance recently, and several XML databases [8–11, 22, 23, 27, 29] 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 [4, 5, 21, 31]. 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 [12] accomplished this, with reasonable success, for relational database

1

systems. However, no benchmark has been successful in the realm of Object-Relational DBMSs (ORDBMSs) and Object-Oriented DBMSs (OODBMSs) 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 [13, 30] provides 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 work 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.

The challenges of a micro-benchmark are different from those of a holistic benchmark. The benchmark data set must be *complex* enough to incorporate data characteristics that are likely to have an impact on the performance of query operations. However, at the same time, the benchmark data set must be *simple* so that it is not only easy to pose and understand queries against the data set, but 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 and element content values, and the cardinality of element nodes, there are several other characteristics, such as tree fanout and tree depth, that contribute to the rich structure of XML data. An XML benchmark must in-

2

corporate 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 work are:

- The specification of 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 ORDBMS.

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

2   Related Work

Several proposals for generating synthetic XML data have been made [1,3]. 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. [3] 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 the synthetic 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.

Several benchmarks (XMach-1, XMark, XOO7, and XBench) [4, 5, 21, 31] have been proposed for evaluating the performance of XML data management systems. XMach-1 [4] and XMark [21] generate XML data that models data from particular Internet applications. In XMach-1 [4], the database contains a directory of all collected XML documents and the management documents. The directory represents structured data, and the managed documents are text documents. In XMark [21], the data is based on an Internet auction application that consists of relatively structured and data-oriented parts. XOO7 [5] is an XML version of the OO7 Benchmark [7], 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 [15,16], Kweelt [19], and an XML-enabled RDBMS. The results of the tests conclude that XML-enabled RDBMSs are efficient

3

in processing queries on data-centric documents while native XML databases are efficient in processing queries on document-centric queries [17]. Recognizing that different applications requires different benchmarks, Yao et al. [31] 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 change of the system performance as 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. Although our benchmark data set is static, different parts of this single data set have different data characteristics. Therefore, the benchmark data set and queries can be used to analyze the performance with respect to different XML data characteristics.

Finally, we note that [20] presents desiderata for an XML database benchmark, identifies key components and operations, and enumerates ten challenges that XML benchmarks should address. The central focus of [20] 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. Since in the future, we may modify the benchmark data set and schema, to avoid confusion, we call this version of the benchmark "Mbench-v1".

3.1 A Discussion of the Data Characteristics

In a relational paradigm, the primary data characteristics that are important from a query evaluation perspective 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 evaluating indirect containment relationships between nodes (which essentially requires determining node pairs that have ancestor-descendant relationships). Similarly, the fanout of nodes can affect the way in which the DBMS stores the data and answers

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

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 a large number of data sets make 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 fanout factor is isolated from other factors, such as the number of nodes. For instance, the number of nodes is the same (2,704) at levels 7 and 9. Nodes at level 7 have a fanout of 13, whereas nodes at level 9 have a fanout of 2. A pair of queries, one against each of these two levels, can be used to isolate the impact of fanout. 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 those 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 the numbers of nodes, depths, or fanouts. We would like to isolate the effect of the number of nodes from the effects of other structural changes, such as depth and fanout. We achieve this by keeping the tree depth constant for all scaled versions of the data set and changing the number of fanouts of nodes at only a few levels, namely levels 5-8. In the design of the benchmark data 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, and 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 numbers of nodes in the data sets **DSx10** and **DSx100** are 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 Base-Type. 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 using [18].

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 values 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.

---

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

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 nested elements. The first is of type BaseType, and the second is of type Occasional-Type. The number of repetitions of a BaseType element is determined by the fanout of the parent element, as described in Figure 1. On the other hand, the number of occurrences of an OccasionalType element is 1 if the aSixtyFour attribute of its BaseType parent element has value 0; otherwise, the number of occurrences is 0. An Occasional-Type element has content that is identical to the content of the parent but has only one attribute, aRef. Every OccasionalType element is a leaf node in the XML data tree, and does not nest any other elements. 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 the aRef attribute of the Occasional-Type element.) In the case where there is no BaseType element that 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 value of its aRef is equal to the value of aUnique1 of the root node).

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 by applying Zipf's law [32]. 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").

---

[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 Document Specification in XML Schema

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

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

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

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

Figure 3: The Template for Generating Strings

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 accurately reflect word occurrence probabilities in a wide variety of situations. The value of the aString attribute is simply the first line of the long string of the element content.

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

An example of an eNest element, with its content, is shown in Figure 4.

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 physical operations, we can estimate the cost of any complex query by just adding up relevant piecewise

```
< eNest aUnique1="368" aUnique2="43719" aLevel="8"
    aFour="3" aSixteen="7" aSixtyFour="7"
    aString="Sing a song of fifteenB5">

    Sing a song of fifteenB5
    A pocket full of 45eightyB15
    Four and twenty fiveB5
    All baked in a 6seventynineB11.

    When the fourB7 was opened,
    The fortyeightB7 began to sing;
    Wasn't that a dainty 72fiftytwoB15
    To set before the 8sixtyfiveB14?

    The King was in his oneB1,
    Counting out his 35elevenB13;
    The Queen was in the sixteenB5
    Eating bread and 10ninetyeightB12.

    The maid was in the 28ninetyoneB15
    Hanging out the 2fiftyB11;
    When down came a 15fortyB14,
    And snipped off her 9sixtyfiveB11!
< /eNest >
```

Figure 4: Sample BaseType Element (shown without any nested sub-elements).

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. Thus, we choose a set of simple queries that can indicate the performance of the simple components, instead of a complex query. However, with the designed data set, an engineer can also define additional queries to test the impact of data characteristics that may affect the performance of their developing engines. For example, an engineer who is interested in the impact of fanout to the system performance can devise a pair of queries: one requests nodes at level 7 and the other one requests nodes at level 9. At levels 7 and 9, the number of nodes is the same, but the number of node fanouts is different.

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 elements (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, or other possibilities, such as returning all the nodes that participated in the match. 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] In this manuscript, the term "selectivity" refers to the percentage of the selected element nodes. Thus, a selectivity of 1.6% for this query implies that there will be one node returned as an output for every 64 nodes in the document.

- **QR1. Only element.** Return only the elements in question, not including any children of the elements
- **QR2. Subtree.** Return all nodes in the entire sub-tree rooted at the elements.

Default Returned Structure: 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. Note that a query with its selectivity value = x% does not return the result that takes about x% of the total bytes of all data since only the identifier attributes of the selected nodes are returned

4.1.2 Selection on Values

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

- **Exact Match**
  QS1. Selective attribute. Select nodes with aString = "Sing a song of oneB4". Selectivity is 0.8%.

---

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

QS2. Non-selective attribute. Select nodes with aString = "Sing a song of oneB1". Selectivity is 6.3%.

Selection on range values.

QS3. Range-value. Select nodes with aSixtyFour between 5 and 8. Selectivity is 6.3%.

- **Approximate Match**

  QS4. Selective word. Select all nodes with element content such that the distance between keyword "oneB5" and keyword "twenty" is at most four. Selectivity is 0.8%.

  QS5. Non-selective word. Select all nodes with element content such that the distance between keyword "oneB2" and keyword "twenty" is at most four. Selectivity is 6.3%.

For both queries QS4 and QS5, the matching condition requires the two matched words to be part of the *same* element content.

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 aUnique1 attribute of the root node of the selection pattern, unless specified otherwise. In these queries, the selectivity of a predicate is noted following the predicate.

- **Order-Sensitive Selection**

  QS6. 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). Overall query selectivity is 3.1%.

  QS7. 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.

- **Parent-Child Selection**

  QS8. Non-selective parent and selective child. Select nodes with aLevel = 15 (sel=23.8%, approx. 1/4) that have a child with aSixtyFour = 3 (sel=1/64). Overall query selectivity is approximately 0.7%.

  QS9. Selective parent and non-selective child. Select nodes with aLevel = 11 (sel=1.5%, approx. 1/64) that have a child with aFour = 3 (sel=1/4). Overall query selectivity is approximately 0.7%.

- **Ancestor-Descendant Selection**

  QS10. Non-selective ancestor and selective descendant. Select nodes with aLevel = 15 (sel=23.8%, approx. 1/4) that have a descendant with aSixtyFour = 3 (sel=1/64). Overall query selectivity is 0.7%.

  QS11. Selective ancestor and non-selective descendant. Select nodes with aLevel = 11 (sel=1.5%, approx. 1/64) that have a descendant with aFour = 3 (sel=1/4). Overall query selectivity is 1.5%.

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

  In the ancestor-descendant queries above (QS10-QS11), ancestors are never nested below other ancestors. To test the performance of queries when ancestors are recursively nested below other ancestors, we have two other ancestor-descendant queries, QS12-QS13. These queries are variants of QS10-QS11.

  QS12. Non-selective ancestor and selective descendant. Select nodes with aFour = 3 (sel=1/4) that have a descendant with aSixtyFour = 3 (sel=1/64).

  QS13. Selective ancestor and non-selective descendant. Select nodes with aSixtyFour = 9 (sel=1/64) that have a descendant with aFour = 3 (sel=1/4).

The overall selectivities of these queries (QS12-QS13) cannot be the same as that of the "equivalent" unnested queries (QS10-QS11) for two situations – first, the same descendants can now have multiple ancestors that 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 5 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.

  The benchmark uses the following complex queries:

  QS14. One chain query with three parent-child joins with the selectivity value 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%). Overall query selectivity is close to 0%.
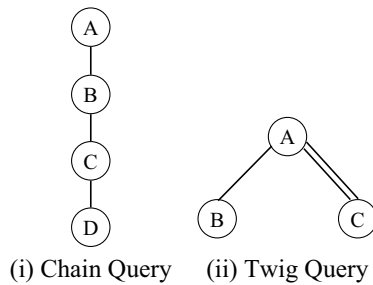
(i) Chain Query   (ii) Twig Query

Figure 5: Samples of Chain and Twig Queries

QS15. One twig query with two parent-child joins with the selectivity value 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). Overall query selectivity is close to 0%.

QS16. One twig query with two parent-child joins with the selectivity value 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). Overall query selectivity is close to 0%.

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

  QS17. Select all BaseType elements below where there is no OccasionalType element. Overall query selectivity is 93.2%.

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 returned structure is the pair of the aUnique1 attributes of nodes joined.

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

QJ2. Non-selective value-based join. Select nodes with aSixteen = 2 (sel=1/16) and join with themselves based on the equality of the 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

14

some XML databases.

QJ3. Selective pointer-based join. Select all OccasionalType nodes that point to a node with aSixtyFour = 3 (sel=1/64). Selectivity is 0.02%.

QJ4. Non-selective pointer-based join. 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 nodes of type OccasionalType, not the nodes pointed to.

4.4   Aggregation

Aggregate queries are very important for many applications, such as 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 with groupby. Compute the average value of the aSixtyFour attribute of all nodes at each level. The returned 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.

QA2. Structural aggregate selection. Select elements that have at least two children that satisfy aFour = 1 (sel=1/4). Selectivity is 3.1%.

QA3. Structural exploration. For each node at level 7 (have aLevel = 7 (sel=0.4%)), determine the height of the sub-tree rooted at this node. Selectivity is 0.4%.

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

4.5   Update and Load

The benchmark also contains three update queries which include insert, delete, and bulk load.

QU1. Insert. Insert a new node below each node with aSixtyFour = 1. Each new node has attributes identical to its parent, except aUnique1, which is set to some new large, unique value, not necessarily contiguous with the values already assigned in the database.

QU2. Delete. Delete all leaf nodes with aSixteen = 3.

QU3. Bulk Load. Load the original data set from a (set of) document(s).

4.6   Document Construction

Many XML applications often require construction and reconstruction of large new documents from the data stored. These applications usually deal with document-centric data, such as Web pages and on-line news.

QC1. Structure preserving. Return a set of documents, one for each sub-tree rooted at the node at level 11 (have aLevel = 11) and that has a child of type OccasionalType.

QC2. Structure transforming. 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. Return a set of documents, one for the sub-tree rooted at each node 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 a native commercial XML DBMS, a university native XML DBMS, and a leading commercial ORDBMS. Due to the nature of the licensing agreement for the commercial systems, we cannot disclose the actual names of the systems, and will refer to the commercial native system as **CNX**, and the commercial ORDBMS as **COR**.

The native XML database Timber [29] is a university native XML DBMS that we are developing at the University of Michigan [29]. Timber uses the Shore storage manager [6], and implements various join algorithms, query size estimation, and query optimization techniques that have been developed for XML DBMSs.

The ORDBMS is provided by a leading database vendor, and we used the Hybrid inlining algorithm to map a DTD into a relational schema [26]. 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 Michigan benchmark's web site [28].

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 XQuery queries on this commercial system. Consequently, we reverted to writing the queries using XPath expressions. In all the cases that we could run queries using the XQuery interface, the XPath approach was faster. Consequently, all the query execution times reported here are for queries written in XPath. 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 Michigan benchmark's web site [28].

5.1 Experimental Platform and Methodology

All experiments were run on a single-processor 550 MHz Pentium III machine with 256 MB of main memory, running Windows 2000. The benchmark machine has 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 systems choose 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 in most cases 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 does not effectively handle the "//" operator, which is invoked often in the benchmark. Furthermore, the index is not effective for retrieving BaseType elements which are 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), which we refer to as **DSx1**. Although we wanted to load larger scaled up data sets, we found that in many cases the XML 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 refer to as **DSx0.1**, 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 of the size of the **DSx1** 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 on 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 Michigan benchmark's web site [28], allows the 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 which shows the 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 6 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 figure, *N/A* indicates that queries could not be run with the given configuration and system software.

Discussed Section	Query Group (Query numbers)	Geometric Mean Response Times (seconds)							
		DSx0.1				DSx1			
		CNX		Timber	COR	CNX		Timber	COR
		Idx	No Idx			Idx	No Idx		
5.2.1	Returned structure (QR1-QR2)	3.16	2.81	0.06	0.13	9.19	7.93	5.57	2.05
5.2.2	Exact match (QS1-QS3)	2.63	2.25	0.03	0.04	7.69	6.77	0.16	0.31
5.2.3	Approxity match (QS4-QS5)	N/A	N/A	3.00	1.12	N/A	N/A	32.92	44.09
5.2.4	Order-sensitive (QS6-QS7)	2.64	2.28	0.00	0.03	7.80	6.75	0.38	0.17
5.2.5	P-C selection (QS8-QS9)	2.75	2.38	0.17	0.05	7.70	6.98	1.76	0.39
5.2.5	A-D selection (QS10-QS11)	3.18	2.65	0.17	0.38	8.75	7.72	1.73	7.48
5.2.5	Ancestor nesting (QS12-QS13)	3.42	2.86	0.17	0.93	9.32	8.39	1.77	10.18
5.2.6	Complex pattern (QS14-QS16)	3.87	3.37	0.28	0.03	7.72	6.90	5.28	0.50
5.2.7	Negated selection (QS17)	3.19	2.84	1.29	2.10	82.06	66.15	12.58	23.38
5.2.8	Value-based join (QJ1-QJ2)	359.14	359.14	1.72	0.05	1247.59	1268.40	18.82	0.42
5.2.8	Pointer-based join (QJ3-QJ4)	163.50	161.79	3.15	0.02	1330.70	1339.54	19.73	0.14
5.2.9	Aggregation (QA1-QA3)	3.03	2.70	2.22	0.57	8.19	7.45	209.84	6.39
5.2.10	Update (QU1-QU2)	N/A	N/A	N/A	1.92	N/A	N/A	N/A	42.79
5.2.10	Bulk load (QU3)	300.00	189.00	192.31	67.00	5418.00	2699.00	2391.46	807.33
5.2.11	Construction (QC1-QC2)	N/A	N/A	N/A	3.39	N/A	N/A	N/A	42.51

Figure 6: Benchmark Numbers for the Three DBMSs

From Figure 6, we observe that Timber is very efficient at processing XML structural queries (QS6-QS17). The implementation of "traditional" relational-style queries such as value-based joins (QJ1-QJ4) is not highly tuned in Timber. This is primarily because Timber is a research prototype and most of the development attention has focused on XML query processing issues that are not covered by traditional relational techniques.

COR can execute almost all queries well, except the ancestor-descendant relationship queries (QS10-QS13). COR performs very well on the parent-child queries (QS8-QS9), which are evaluated using foreign key joins.

From Figure 6, we observe that CNX is usually slower than the other two systems. A large query overhead, of about 2 seconds on the DSx0.1 data set, is incurred by

CNX, even for small queries. (This overhead was measured by issuing a query that only retrieves the single root element of the document.) CNX is also considerably slower than Timber.

We suspect that the reason for the poor performance of CNX is fourfold. First, although we had structure indices built in 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 indices for its structural indices. As a result, since all elements in the benchmark data set are either the eNest or eOccasional elements, tag indices are essentially useless. Furthermore, when the system is forced to follow the tag indices, random accesses occur and result in compromised performance. Third, the value indices on the attributes of the eNest nodes may cause random disk accesses at query evaluation time if the document does not fit entirely in memory, which incurs much higher disk access costs than the sequential scan in the non-indexed database. Finally, in the above situation, the optimizer ideally should have been able to evaluate different query plans and choose the optimal plan, which in this case should be the sequential scan plan. The current version of the CNX optimizer favors index plans even when sequential scan plans are cheaper.

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-QR2)

Figure 7 shows the execution times of the returned structure queries, QR1-QR2.

Query	Query Description	Sel. (%)	Response Times (seconds)							
			DSx0.1				DSx1			
			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 entire sub-tree	1.6	3.97	3.63	0.26	1.09	11.36	10.17	387.23	26.09

Figure 7: Benchmark Numbers of Returned Structure Queries

Examining the performance of the returned structure queries, QR1-QR2, in Figure 7, we observe that the returned structure has an impact on all systems. Timber and COR perform poorly when the whole sub-tree is returned (QR2). 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 [6] 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 QR2 helped

Timber designers identify this implementation weakness, and begin taking steps to address it.

COR takes more time in selecting and returning descendant nodes (QR2) because COR needs to call recursive SQL statements in retrieving the descendant nodes.

CNX also takes more time in returning the entire sub-tree more than just returning the element itself, but the performance difference between returning only the element and returning the element with its sub-tree is not dramatic as in Timber and COR.

Selection on Values (QS1-QS3)

In this section, we examine the performance of the three systems for the selection on values queries. The performance numbers are shown in Figure 8.

Query	Query Description	Sel. (%)	Response Times (seconds)							
			DSx0.1				DSx1			
			CNX		Timber	COR	CNX		Timber	COR
			Idx	No Idx			Idx	No Idx		
QS1	Selection on string value (selective)	0.8	2.36	1.99	0.03	0.02	6.96	6.06	0.05	0.08
QS2	Selection on string value (non-sel.)	6.3	2.40	2.05	0.03	0.06	7.08	6.21	0.34	0.63
QS3	Selection on range values	6.3	3.21	2.81	0.03	0.06	9.24	8.23	0.23	0.60

Figure 8: Benchmark Numbers of Selection on Values Queries

5.2.2 Exact Match (QS1-QS3)

Single Attribute Selection (QS1-QS2)
Out of the three tested systems, selectivity has the most impact on COR. The response time of the non-selective query (QS2) is more than that of the selective query (QS1), with the response times grow linearly with increasing selectivity. On the other hand, selectivity has an impact on Timber only on a large data set (**DSx1**), and not on the relatively smaller data set (**DSx0.1**).

Overall, CNX does not perform as well as the other two DBMSs. Although selectivity has impact on CNX, it is not as strong as with the other two DBMSs (this is true even with indexing in CNX). Although the response time of the non-selective query (QS2) is higher than that of the selective query (QS1), the difference does not reflect a linear growth.

Range Selection (QS3)

Both Timber and COR handle the range predicate query just as well as the equality predicate. In both systems, the performance of the range predicate query (QS3) is almost the same as that of the comparable equality selection query (QS2). On the

other hand, CNX takes a little more time to evaluate the range predicate query than to evaluate the comparable equality selection query.

5.2.3 Approximate Match (QS4-QS5)

Figure 9 shows the execution times of the approximate match queries, QS4-QS5.

Query	Query Description	Sel. (%)	Response Times (seconds)							
			DSx0.1				DSx1			
			CNX		Timber	COR	CNX		Timber	COR
			Idx	No Idx			Idx	No Idx		
QS4	String distance selection (selective)	0.8	N/A	N/A	2.49	0.95	N/A	N/A	27.23	42.79
QS5	String distance selection (non-sel.)	6.3	N/A	N/A	3.61	1.31	N/A	N/A	39.79	45.42

Figure 9: Benchmark Numbers of Approximate Match Queries

In COR, to measure the distance between words contained in a long string (QS4-QS5), we need to invoke a user-defined function, which cannot make use of an index. Consequently, the efficiency of the query is independent of the selectivity of the string distance selection predicate. CNX currently does not support string distance selections.

Structural Selection (QS6-QS17)

5.2.4 Order-sensitive Selection (QS6-QS7)

The execution times of the order selection queries, QS6-QS7, are shown in Figure 10.

Query	Query Description	Sel. (%)	Response Times (seconds)							
			DSx0.1				DSx1			
			CNX		Timber	COR	CNX		Timber	COR
			Idx	No Idx			Idx	No Idx		
QS6	Local ordering	3.1	2.73	2.37	1.45	0.08	8.31	7.14	14.58	1.02
QS7	Global ordering	1 node	2.56	2.19	0.00	0.01	7.32	6.39	0.01	0.03

Figure 10: Benchmark Numbers of Order-sensitive Queries

In CNX, local ordering (QS6) and global ordering (QS7) are slightly different from each other.

In Timber, local ordering (QS6) results in considerably worse performance than global ordering (QS7) because it requires many random accesses. On the other hand, global ordering (QS7) performs well because it requires only one random access.

In COR, local ordering (QS6) is more expensive than global ordering (QS7). This is because local ordering (QS6) needs to access a number of nodes that satisfy the

given order. On the other hand, QS7 quickly returns as soon as it finds the first tuple that satisfies the given order and predicates.

5.2.5 Simple Containment Selection (QS8-QS13)

Figure 11 shows the performance of selected structural selection queries, QS8-QS17. In this figure, "P-C:low-high" refers to the join between a parent with low selectivity value and a child with high selectivity value, whereas, "A-D:high-low" refers to the join between an ancestor with high selectivity value and a descendant with low selectivity value.

Query	Query Description	Sel. (%)	Response Times (seconds)							
			DSx0.1				DSx1			
			CNX		Timber	COR	CNX		Timber	COR
			Idx	No Idx			Idx	No Idx		
QS8	P-C: high-low	0.7	2.92	2.55	0.17	0.05	8.10	7.40	1.79	0.44
QS9	P-C: low-high	0.7	2.59	2.23	0.17	0.05	7.32	6.58	1.73	0.34
QS10	A-D:high-low	0.7	3.11	2.57	0.18	0.06	8.44	7.40	1.72	2.60
QS11	A-D:low-high	1.5	3.26	2.73	0.16	2.44	9.07	8.06	1.74	21.54
QS12	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
QS13	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
QS14	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
QS15	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
QS16	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
QS17	Negated selection	93.2	3.19	2.84	1.29	2.10	82.06	66.15	12.58	23.38

Figure 11: Benchmark Numbers of Structural Selection Queries

As seen from the results for the direct containment queries (queries QS8 and QS9 in Figure 11), COR processes direct containment queries better than Timber, but Timber handles indirect containment queries (QS10-QS13) better.

CNX underperforms on direct containment queries (QS8-QS9) as compared to the other systems. However, on indirect containment queries (QS10-QS13), it often performs better than COR. CNX only has slightly better performance on direct containment queries (QS8-QS9) than on indirect containment queries (QS10-QS13). Examining the effect of query selectivities on CNX query execution (for example, see queries QS8-QS11), we notice that the execution times are relatively immune to the query selectivities, implying that the system does not effectively exploit the difference in query selectivities when choosing query plans. We also notice that for QS16, the response times on the **DSx0.1** data set are slightly more than those on the **DSx1** data set. We have examined the query plans for both data sets in CNX, and see that the system is

using the same query plan. The small increase is probably noise, and in this case the query execution cost is likely being dominated by overhead costs in executing queries.

In Timber, structural joins [2] 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 evaluating the ancestor-descendant queries is not necessarily higher than that of evaluating the parent-child queries.

(As an interesting side note, when running these structural join queries in Timber, we noticed that the execution time for the structural join algorithms in Timber were more expensive than what we expected. On a closer examination, we found that the memory management of the stack elements used in the structural join algorithms was using repeated allocation and deallocation of fixed sized memory blocks. We then wrote a simple memory manager for fixed-size memory block management, and found that the performance of the structural join algorithms speeded up by as much as a factor of two in the cases where the cost of the join was dominated by the CPU costs.)

COR is very efficient for processing parent-child queries (QS8-QS9), since these translate into foreign key joins which the system can evaluate efficiently using indices. On the other hand, COR has much longer response times for the ancestor-descendant queries (QS10-QS13).

For evaluating A-D queries in COR, we tried two approaches: the first approach is to call recursive SQL statements, and the second approach is to write SQL queries and then combine the results of these queries using union operators. We note that for A-D queries when the schema of data is not available, only the first approach is applicable. However, in our benchmark the schema is available, which allows the system to used either of these two approaches. In this paper, we report the best execution time of each query when using these two approaches. We found that using SQL queries and union operators approach was suitable for queries on data without ancestor nesting or queries on very shallow data. As an example, for query QS10 on **DSx1** data set, when using a recursive SQL statement, the response time is 21.54 seconds. In contrast, for this query using the second approach resulted in a query evaluation time of 2.6 seconds. The recursive approach is suitable for queries with ancestor-descendant queries on deeply nested data. For example, the cost of evaluating query QS12 on **DSx1** data set using multiple SQL queries and union operators is 334.86 seconds. In contrast, the execution time for this query is only 8.83 seconds using a recursive SQL statement.

All systems, except COR, are immune to the recursive nesting of ancestor nodes below other ancestor nodes; the queries on recursively nested ancestor nodes (QS12-QS13) have the same response times as their non-recursive counterparts (QS10-QS11).

5.2.6 Complex Pattern Containment Selection (QS14-QS16)

Overall, Timber performs well on complex queries. It breaks the chain pattern queries (QS14) or twig queries (QS15 and QS16) into a series of binary containment joins, evaluating the most selective joins first.

COR also performs well on these queries because the availability of indices that speed up the foreign key joins between parents and children. 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. However, CNX has an inefficient implementation of the structural indexing, which causes the system to chase each level of nesting in order to find the nodes in question. The index access is likely to take more time than the sequential scan that occurs in the non-indexed database.

5.2.7 Irregular Structure (QS17)

Since some parts of an XML document may have irregular data structures, queries looking for missing elements, such QS17, are useful for checking the capability of a system in handling with irregularities. Query QS17 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 does not scale very well, as it did for other queries. The selectivity of query QS17 is fairly high (93.2%), and as the database size increases, the size of the returned result grows dramatically. CNX spends a large part of its execution time in processing the results at the client, and this part does not scale very well.

In Timber, QS17 is executed very efficiently as this query is evaluated using a variation of the structural join algorithm. This variation of the structural join algorithm executes the usual algorithm for A-D joins, and simply 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 **DSx1** data set). This approach is expensive 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. In this alternative, first we create a view that selects all BaseType elements that have some OccasionalType descendants (this requires a recursive SQL statement). Then a left-outer join query between the view and the relation that holds all BaseType elements is used to select 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 smaller response time of 23.38 seconds, which is reported in Figure 11.

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

The execution times of the join queries, QJ1-QJ4, are shown in Figure 12.

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. Timber performs poorly on both QJ1 and QJ2 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 joins are evaluated using a naive nested loop join algorithm. Thus, the complexity of the execution time of the queries is $O(n^2)$ where n is the data size. The selectivity factor has much impact on the value-based joins in CNX, but not on the pointer-based joins. Notice that CNX scales up better than Timber on QJ1-QJ2.

Query	Query Description	Sel. (%)	Response Times (seconds)							
			DSx0.1				DSx1			
			CNX		Timber	COR	CNX		Timber	COR
			Idx	No Idx			Idx	No Idx		
QJ1	Value-based join (selective)	1.6	188.88	187.50	0.69	0.03	1247.59	1268.40	18.82	0.21
QJ2	Value-based join (non-sel.)	6.3	682.87	687.90	4.29	0.08	> 1 hr	> 1 hr	> 1 hr	0.83
QJ3	Pointer-based join (selective)	0.02	161.50	160.09	0.73	0.01	1307.60	1320.52	20.08	0.05
QJ4	Pointer-based join (non-sel.)	0.4	165.52	163.50	13.63	0.05	1354.20	1358.83	19.38	0.42

Figure 12: Benchmark Numbers of Traditional Join Queries

Query	Query Description	Sel. (%)	Response Times (seconds)							
			DSx0.1				DSx1			
			CNX		Timber	COR	CNX		Timber	COR
			Idx	No Idx			Idx	No Idx		
QA1	Value aggregation with groupby	16 nodes	N/A	N/A	10.11	0.06	N/A	N/A	N/A	0.54
QA2	Structural aggregate selection	3.1 nodes	3.03	2.70	15.38	0.26	8.19	7.45	1288.67	3.65
QA3	Structural exploration	0.4	N/A	N/A	0.07	12.00	N/A	N/A	34.17	132.59

Figure 13: Benchmark Numbers of Aggregate Queries

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 performs worse than 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 that are very efficiently implemented in traditional commercial database systems.

5.2.9   Aggregation (QA1-QA3)

Figure 13 shows the execution times of the aggregation queries, QA1-QA3.

In Timber, the structure of the XML data is maintained and reflected throughout the system. Therefore, a structural exploration query, such as QA3, performs well. On the other hand, a value aggregation query, such as QA1, performs worse due to a large number of random accesses required in fetching the element values, which are not kept clustered.

Query	Query Description	Response Times (seconds)							
		DSx0.1				DSx1			
		CNX		Timber	COR	CNX		Timber	COR
		Idx	No Idx			Idx	No Idx		
QU1	Insert	N/A	N/A	N/A	4.67	N/A	N/A	N/A	41.84
QU2	Delete	N/A	N/A	N/A	0.79	N/A	N/A	N/A	43.76
QU3	Bulk load	300.00	189.00	192.31	67.00	5418.00	2699.00	2391.46	807.33

Figure 14: Benchmark Numbers of Update Queries

Query	Query Description	Response Times (seconds)							
		DSx0.1				DSx1			
		CNX		Timber	COR	CNX		Timber	COR
		Idx	No Idx			Idx	No Idx		
QC1	Structure preserving	N/A	N/A	N/A	0.83	N/A	N/A	N/A	2.48
QC2	Structure transforming	N/A	N/A	N/A	13.81	N/A	N/A	N/A	728.61

Figure 15: Benchmark Numbers of Document Construction Queries

In COR, evaluating the structural aggregation query is much more expensive than evaluating the value aggregation query. 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 accessed quickly using indices.

Since CNX is a native XML database, one would expect it to perform reasonably well on structural aggregation queries. However, it currently does not evaluate the structural aggregation queries as fast as COR. Moreover, it does not support several functionality features needed for aggregation queries, such as QA1 and QA3.

5.2.10 Update (QU1-QU3)

Figure 14 shows the execution times of the update queries, QU1-QU3.

Update operations (QU1 and QU2) are not currently supported in CNX and Timber.

For the bulk loading query QU3, COR is much more efficient than CNX and Timber. This is not surprising as commercial relational system pay careful attention to the performance of their bulk loading algorithms. For example, commercial relational bulk load utilities often bulk load an index by building the index bottom up. In contrast, in Timber we insert each entry into the index one at a time.

5.2.11 Document Construction (QC1-QC2)

Figure 15 shows the execution times of the queries to construct XML documents, QC1-QC2.

The execution time of query QC1 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 (QC2)

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.

Structure preserving and structure transforming operations are currently 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 **DSx0.1** to **DSx1**. For this discussion, we will refer to the results presented in Figure 6.

5.3.1 Scaling Performance on CNX

In almost all the queries, the ratios of the response times when using **DSx0.1** over **DSx1** is at most 10, except for QS17, which consists of a nested aggregate count() function. This indicates that CNX scales at least linearly, and sometimes super-linearly with respect to the database size.

5.3.2 Scaling Performance on Timber

Timber scales linearly for all queries, with a response time ratio of approximately 10, with two exceptions. The first exception is when large returned results have to be constructed. In this case, Timber is inefficient, and scales poorly, as discussed above in Section 5.2.1. The second exception is when the value-based join is invoked. The value-based join implementation is naive, and scales poorly.

5.3.3 Scaling Performance on COR

In COR, the ratios of the response times when using **DSx0.1** over **DSx1** data sets are approximately 10, showing linear scale-up. Exceptions to this linear-scale up occur in three types of queries: a) the approximate match queries, b) the update queries, and c) the document construction queries.

Approximate match queries scale poorly. For queries QS4 and QS5, COR requests a table scan and a user-defined function to search for tuples that satisfy with the predicates. The costs of finding predicate words and computing the distance between words increase rapidly as the table size increases.

Figure 14 indicates that the performance gets worse as the 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.

COR performs poorly on document construction queries. The experimental results of queries, such as QC2, show that COR spends a significant amount of time to reconstruct the results. Recently, Shanmugasundaram et al. [24, 25] have addressed this problem as they proposed techniques for efficiently publishing and querying XML view of relational data. However, these techniques are not currently implemented in COR.

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 ORDBMS. 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 native XML database Timber, 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 statements. While this has been shown to be true for some XML queries in the past [14, 25], we show that this is also true for simple indirect containment queries, and queries that search for irregular structures. We also demonstrate that one can use recursive SQL statements to evaluate any structural query in the benchmark; however, using recursive SQL statements in the ORDBMS is much more expensive than using efficient XML structural join algorithms implemented in Timber.

Finally, we note that the proposed 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. The 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 [28]. It is *scalable* through the use of a scaling parameter. It is *simple* since it comprises only a single document and a set of simple queries, each designed to test a distinct functionality.

References

[1] A. Aboulnaga, J. Naughton, and C. Zhang. Generating Synthetic Complex-structured XML Data. In *Proceedings of the International Workshop on the Web and Databases*, pages 79–84, Santa Barbara, California, May 2001.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Strucutral Joins: A Primitive for Efficient XML Query Processing Pattern Matching. In *Proceedings of the IEEE International on Data Engineering*, San Jose, CA, 2002.

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

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

[5] S. Bressan, M. L. Lee, Y. G. Li, Z. Lacroix, and U. Nambiar. XML Management System Benchmarks. In A. B. Chaudhri, A. Rashid, and R. Zicari, editors, *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison Wesley, 2003.

[6] 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.

[7] M. J. Carey, 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.

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

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

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

[11] Sonic Software Corporation. Sonic XML Server. `http://www.sonicsoftware.com/support/lifecycle/index.ssp#sonic_xml`.

[12] Transaction Processing Performance Council. TPC Benchmarks. `http://www.tpc.org/`.

[13] 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.

[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 *Proceedings of the International Workshop on the Web and Databases*, pages 25–30, Philadelphia, Pennsylvania, June 1999.

[16] 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.

[17] U. Nambiar, Z. Lacroix, S. Bressan, M. L. Lee, and Y. G. Li. Efficient XML Data Management: An Analysis. In *Proceedings of the 3rd International Conference on Electronic Commerce and Web Technologies (ECWeb)*, pages 87–98, Aix en Provence, France, September 2002.

[18] S. K. Park and K. W. Miller. Random Number Generators: Good Ones are hard to Find. *Communications of the ACM*, 31(10), October 1998.

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

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

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

[22] H. Schoning. Tamino - A DBMS Designed for XML. In *Proceedings of the IEEE International on Data Engineering*, pages 149–154, Heidelberg, Germany, April 2001.

[23] H. Schoning and J. Wasch. Tamino - An Internet Database System. In *Proceedings of the International Conference on Extending Database Technology*, pages 383–387, Konstanz, Germany, March 2000.

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

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

[26] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Database for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the International Conference Very Large Data Bases*, pages 302–314, Edinburgh, Scotland, September 1999.

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

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

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

[30] C. Turbyfill, C. U. Orji, and D. Bitton. ASAP - An ANSI SQL Standard Scaleable and Portable Benchmark for Relational Database Sytems. In J. Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann, second edition, 1993.

[31] B. B. Yao, M. Tamer Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proceedings of the IEEE International on Data Engineering*, page To appear, Boston, MA, March 2004.

[32] G. K. Zipf. *Human Behavior and the Principle of Least-Effort*. Addison-Wesley, Cambridge, MA, 1949.

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 $1/16 \times 1 = 1/16$. Since there are eight words in the fourth bucket (2^{4-1}), the probability of "oneB4" being picked is $1/16 \times 1/8 = 1/128$.

QR1-QR2. 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 aSixtyFour between 5 and 8. Selectivity is $4 \times 1/64 = 1/16(6.3\%)$.

QS4. 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%).

QS5. 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%).

QS6. Select the second element below each element with aFour = 1 if that second element also has aFour = 1. Let n_l be the number of nodes at level l and f_{l-1} be the fanout at level $l - 1$. Then, the selectivity for "second element" nodes is $\sum_{l=2}^{l=16}(n_l/f_{l-1})/\sum n_l \approx 1/2$. Since the selectivity of the element with aFour =1 is 1/4, the 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%).

QS7. Select the second element with aFour = 1 below any element with aSixtyFour = 1. This query returns at most one element.

QS8. 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%.

QS9. 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%.

QS10. 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 satisfies the second predicate is $(1 - (1/64))^2$. The overall selectivity is $0.24 \times (1 - (1 - (1/64))^2) = 0.7\%$.

QS11. 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 $1.5 \times (1 - (1 - (1/4))^{62}) = 1.5\%$.

QS14. 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 $1/4 \times 1/16 \times 1/16 \times 0.47 = 0.0\%$.

QS15. 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 $0.015 \times 1/4 \times 1/64 = 0.0\%$.

QS16. 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 $0.25 \times 0.015 \times 1/64 = 0.0\%$.

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 each level. This query returns 16 nodes which contains the average values for 16 levels.

QA2. 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%)

QA3. 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%.