XMark: A Benchmark for XML Data Management

Albrecht Schmidt[1] Florian Waas[2] Martin Kersten[1] Michael J. Carey[3]
Ioana Manolescu[4] Ralph Busse[5]

[1] CWI, Kruislaan 413, 1090 GB Amsterdam, The Netherlands, *firstname.lastname*@cwi.nl
[2] Microsoft Corporation, Redmond (WA), USA, florianw@microsoft.com
[3] BEA Systems, Inc., USA, mcarey@bea.com
[4] INRIA-Rocquencourt, 78153 Le Chesnay Cedex, France, Ioana.Manolescu@inria.fr
[5] FHG-IPSI, Dolivostr. 15, 64293 Darmstadt, Germany, busse@ipsi.fraunhofer.de

Abstract

While standardization efforts for XML query languages have been progressing, researchers and users increasingly focus on the database technology that has to deliver on the new challenges that the abundance of XML documents poses to data management: validation, performance evaluation and optimization of XML query processors are the upcoming issues. Following a long tradition in database research, we provide a framework to assess the abilities of an XML database to cope with a broad range of different query types typically encountered in real-world scenarios. The benchmark can help both implementors and users to compare XML databases in a standardized application scenario. To this end, we offer a set of queries where each query is intended to challenge a particular aspect of the query processor. The overall workload we propose consists of a scalable document database and a concise, yet comprehensive set of queries which covers the major aspects of XML query processing ranging from textual features to data analysis queries and *ad hoc* queries. We complement our research with results we obtained from running the benchmark on several XML database platforms. These results are intended to give a first baseline and illustrate the state of the art.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

1 Introduction

The data exchange format XML has been penetrating virtually all areas of Internet application programming. Thus, electronic commerce sites and content providers who rely heavily on the new technology are increasingly interested in deploying advanced data management systems for sites whose data volume exceeds toy sizes. The complexity of the challenge has also attracted the attention of the database research community. Early efforts mainly concentrated on schema issues and the theory of organizing data without a fixed schema, which first seemed incompatible with existing technology. However, as XML gained more and more momentum and numerous commercial products have been appearing on the market – many more being under development – the focus of research shifted; specific technical issues like physical data breakdown and query performance have started to determine the success or failure of implemented XML-based solutions.

Increasingly, major and minor database vendors (see [4] for a seemingly ever-growing list) are scrambling to leverage their existing products – well beyond the rudimentary XML support like conversion of purely relational data to XML documents which most products already provide – with whatever one may need to meet the new requirements. However, these new requirements are still somewhat sketchy and, though the differences between XML and relational or object-relational data are easy to grasp, the implications on the underlying data store are not fully understood yet.

XML, by definition, is a textual markup language which means that unlike relations in (O)RDBMS, data elements are ordered by nature; *string* is the core data type, from which richer data types, *e.g.*, integers, floats and even user-defined abstract data types are derived. Externally provided schema information, which may or may not be present, can help to avoid excessive and expensive coercions between data types. Additionally, to cope with the tree structure of XML documents and the resulting intricate hierarchical relationships between data, regular path expressions are an essential ingredient of query languages and hence call for efficient evaluation strategies. References are a powerful language feature to model relationships that exceed the limitations of tree structures and require further mapping logics like logical OIDs or join indexes for efficient management.

Earlier work on how to extend existing data models to cope with the new XML requirements provided helpful guidance; but since almost all of these prototypes were implemented *on top* of data or object stores, *i.e.*, using standard APIs but without direct access to the internal workings of a product, the conclusions drawn are only valid to a certain extent and the effectiveness of a particular mapping remains unclear. Often, simple extensions to the product could have caused significant performance improvements [17]. Due to their complexity and interdependencies with various system components, most of the designs are hard to assess without putting them to the only conclusive test: a comprehensive quantitative assessment, or in short the right benchmark.

The need for new benchmarks has been a recurring momentum in database research, and so, over the past years the database community developed a rich tradition in performance assessment of systems ranging from research developments like the Hypermodel [1], OO1-Benchmark [10], OO7-Benchmark [8] or the BUCKY benchmark [9] to industrial standards like the family of TPC benchmarks [15] just to mention a few examples. However, none of the available benchmarks offers the coverage needed for XML processing: all of them are geared towards a certain data model but the flexibility and expressiveness of semi-structured query languages exceed existing systems' limitations by far.

The XMark Benchmark described in this paper takes on the challenge and features a tool kit for evaluating the retrieval performance of XML stores and query processors: a workload specification, a scalable benchmark document and a comprehensive set of queries, which were designed to feature natural and intuitive semantics. To facilitate analysis and interpretation, each of the queries is intended to challenge the query processor with an important primitive of the query language. This is useful in a number of ways. In the first place, a systematic examination of the query processors proves beneficial as a such a processor can operate on a variety of architectures, each of which tends to be suited for different application workloads and exhibits special characteristics. For instance, XML stores have been derived from relational, object-relational, main-memory and object-oriented database technology as well as from textual information retrieval data structures and persistent object stores. Therefore, different products can be expected to display diverging behavior in performance and stress tests according to their system architecture and physical data breakdown. Second, the benchmark document and the queries can aid in the verification of query processors, which has been a challenging problem since high level query languages were introduced [24]. In the world of XML, the problem of equivalence of query processor output goes from bad to worse: the degrees of freedom that the different possible physical representations of the document (see [5] for an attempt to tackle it) introduce, are combined with the degrees of freedom in query execution with regards to the order of set-valued attributes, different character encodings, namespaces *etc.* Our experience suggests that the problem of deciding when to regard the output of XML query processors as equivalent still requires research. Third, executing the benchmark query set exhibits details of the work required to incorporate the query processor into an application scenario. Consequently, the benchmark can also help users to estimate the costs of actually deploying such a system in their application scenario and to answer the question what systems fits best their needs.

XML processing systems usually consist of various logical layers and can be physically distributed over a network. To make the benchmark results interpretable we abstract from the systems engineering issues and concentrate only on the core ingredients: the query processor and its interaction with the data store. We do not consider network overhead, communication costs (*e.g.*, RMI, HTTP, CORBA, Sockets, RPC, Java Beans, or others) or transformations (*e.g.*, XSLT) of the output. As for the choice of language, we use XQuery [11], an amalgamation of many research languages for semi-structured data or XML (see [3] for an overview) and a proposed standard. It is in the process of standardization as the language of choice of the major competitors in the field. We do not consider updates other than bulkload as there is little agreement on semantics and a standard is yet to be defined.

The target audience of this paper can be divided into three groups. First, the framework presented here can help database vendors to verify and refine their query processors by comparing them to other implementations. Second, customers can be assisted in choosing between products by using our setting as a simple case study or pilot project that yet provides essential ingredients of the targeted system. For researchers, lastly, we provide example data and a framework for helping to tailor existing technology for use in XML settings and for refinement or design of algorithms.

The rest of the paper is structured as follows: First, we motivate the necessity of a benchmark for XML query processors, then we introduce the structure of the document database. After presenting the queries we give some results and interpretations obtained by running the queries in our test environments.

2 XML Query Processing

Existing database benchmarks cover a plethora of aspects of traditional data management ranging from query optimization to transaction processing. But even if we make use of established techniques to store and process XML, it is not clear if and in what way the semi-structured nature of the data impacts on performance and engineering issues; it might impede on the effectiveness that these techniques have in their original area. In the sequel, we motivate the need for a new benchmark specifically for XML query processors.

The evolution of XML differs significantly from the evolution of relational databases in that for XML there was an agreed-upon standard at an early stage which was accepted and supported by a large community. This imposes a top-down perspective for the benchmark designer resulting in a kind of thematic benchmark, in the sense that it should provide challenges for typical query primitives. Thus, the combination of traditional and new features present in XML processing systems results in the need for a new quality of systems engineering and, hence, a new benchmark.

While it has been shown that many 'data-centric' documents, *i.e.*, documents which logically represent data structures [4], map very nicely to relational databases (*e.g.*, see [14, 20, 23]) or object-relational databases [16], it is less clear how the same systems can handle efficiently documents that are 'document-centric', *i.e.*, natural language with mark-up interspersed. Therefore we want to hint at how different DBMS architectures respond to the XML challenge, which can be summarized as follows: (1) The textual *order* of XML structures as in the original document can be incorporated into queries: a feature that can make simple look-up queries expensive in systems that are not prepared for this challenge (see Queries 2 and 3 in Section 6). (2) Strings are the basic data type; they can vary greatly in length posing additional storage problems. *Type problems* can also arise as the typing rules of query languages tend to clash the generic string tokens of XML. (3) Queries involving *hierarchical structures* in the form of complicated path expressions, especially $1 : n$ relationships in connection with order being queried tend to require expensive join and aggregation operations when executed on relational systems. (4) What compounds matters is the *loose schema* of many XML documents which tends to make query formulation tedious from a user's point of view. Technically, NULL values can blow up the size of the database. Also, specifying long and complicated path expressions is error-prone.

Activities in the context of XML Schema [25] try to allay some of these challenges by making data-centric documents more accessible for (O)RDBMS by reformulating concepts integrity constraints in an XML context. This can indeed solve many problems but requires additional engineering efforts and thus sacrifices some of the quick-and-easy-to-use appeal that helped XML gain popularity so quickly. The benchmark queries have been designed to address these matters specifically.

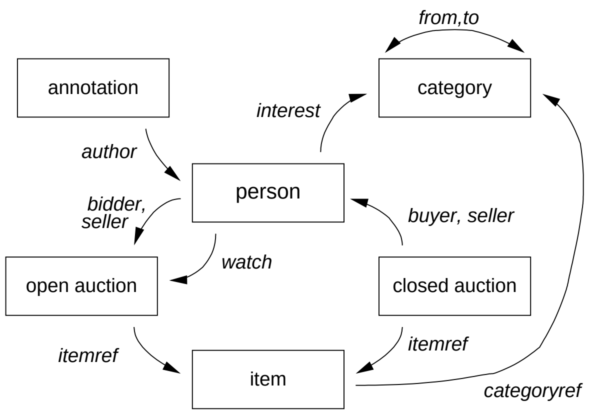

Figure 2: References

3 Related Work

By now there are other benchmarks available that can be used to evaluate certain aspects of XML repositories or database systems. XMach-1 [2] is a benchmark developed at the University of Leipzig. It consists of eight queries and three update operations. The goal of the benchmark is to test how many queries per second a database can process at what cost. Additional measures include response times, bulk load times and database or index sizes. The main objective of the benchmark is to stress-test XML systems under a multi-user workload. XOO7 [7] is the XML counterpart of the OO7 benchmark [8], which is geared towards object repositories; it comprises an XML version of the original OO7 database against which reformulations of the original queries are run; accordingly, the challenges are traversal-oriented. The benchmark also features extensions which are tailored towards testing XML-specific features. Our work differs from them in that it aims at large-scale analytical XML processing (unlike [7]) and at the same time offers query challenges that are designed along the lines of XML query algebras (unlike [2]) thus helping to analyze and improve the underlying query processor rather than merely measuring systems performance.

4 Database Description

The design of an XML Benchmark requires a cautiously modeled example database to make the behavior of queries predictable and to allow for the formulation of queries that both feel natural and present concise challenges. Let us first outline the characteristics of the document and then have a closer look at the technical issues of generating such documents.

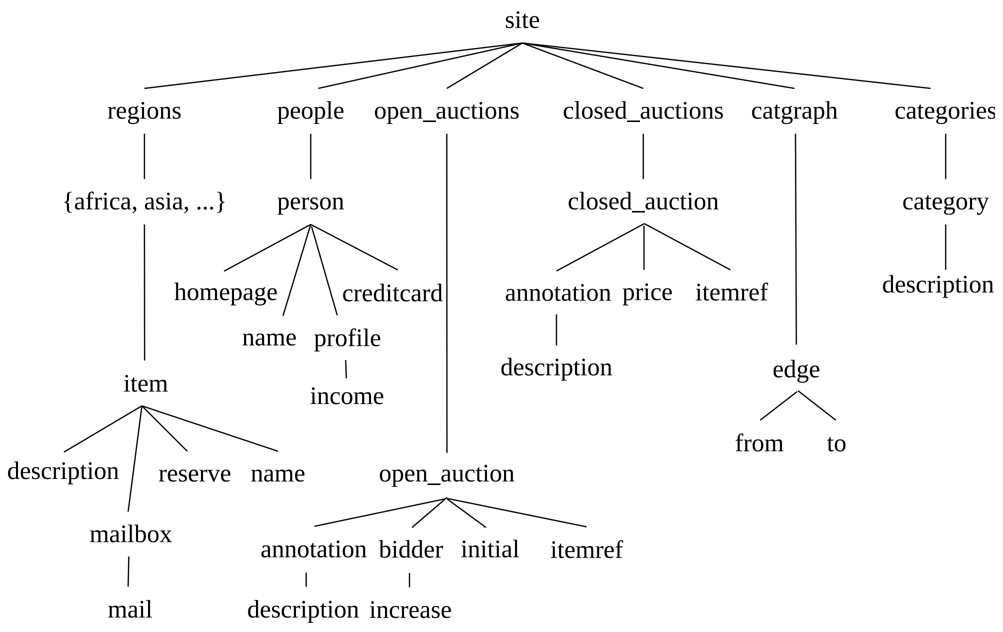

Figure 1: Element relationships between most of the queried elements

4.1 Hierarchical Element Structure

The nesting of elements renders the overall tree structure of XML documents. In this subsection we describe the structure of the benchmark document, which is modeled after a database as deployed by an Internet auction site. The main entities come in two groups: *person*, *open auction*, *closed auction*, *item*, and *category* on the one side and *annotation* and *description* on the other side. The relationships between the entities in the first group are expressed through references while those of the second group, which take after natural language text and are document-centric element structures, are embedded into the sub-trees to which they semantically belong. The hierarchical schema is depicted in Figure 1; an ER diagram can be found in [7]. The semantics of the entities just mentioned is as follows: (1) *Items* are the objects that are on for sale or that already have been sold. Each *item* carries a unique identifier and bears properties like payment (credit card, money order, ...), a reference to the seller, a description *etc.*, all encoded as elements. Each item is assigned a world region represented by the item's parent. (2) *Open auctions* are auctions in progress. Their properties are the privacy status, the bid history (*i.e.*, increases or decreases over time) along with references to the bidders and the seller, the current bid, the time interval within which bids are accepted, the status of the transaction and a reference to the item being sold, among others. (3) *Closed auctions* are auctions that are finished. Their properties are the seller (a reference to a person), the buyer (a reference to a person), a reference to the respective item, the price, the number of items sold, the date when the transaction was closed, the type of transaction, and the annotations that were made before, during and after the bidding process. (4) *Persons* are characterized by name, email address, phone number, mail address, homepage URL, credit card number, profile of their interests, and the (possibly empty) set of open auctions they are interested in and get notifications about. (5) *Categories* feature a name and a description; they are used to implement a classification scheme of *items*. A *category_graph* links categories into a network.

These entities constitute the relatively structured and data-oriented part of the document: the schema is regular on a per entity basis and exceptions, such as that not every person has a homepage, are predictable. Apart from occasional list types such as bidding histories, the order of the input is not particularly relevant. On the other hand, the offspring of *annotation* and *description* elements makes up the document-centric side of the document. Here the length of strings and the internal structure of sub-elements varies greatly. The markup now comprises itemized lists, keywords, and even formatting instructions and character data, imitating the characteristics of natural language texts. This ensures that the database covers the full range of XML instance incarnations, from marked-up data structures to traditional prose. At [19] we have made available some snippets of the benchmark document.

4.2 References

An overview of the references that connect sub-trees is given in Figure 2. Care has been taken that the references feature diverse distributions, derived from uniformly, normally and exponentially distributed random variables. Also note that all references are typed, *i.e.*, all instances of an XML element point to the same type of XML element; for example, references that model interests always refer to categories.

4.3 Generated Text

To generate text that bears similarities with natural language, we analyzed Shakespeare's plays and determined statistic characteristics like word frequency, stopwords *etc*. The generator mimics these characteristics by using the 17000 most frequent words excluding stop words. We did not incorporate additional characteristics like punctuation as they are only of little relevance for the performance assessment. We believe that tokenization and other text compression methods commonly used can be sufficiently well assessed with the text we provide. For entities like names, email addresses *etc.* we used various Internet sources like electronically available phone directories and scrambled them. We refer to [22] for more details.

4.4 XML Constructs

The XML Standard [6] defines constructs that are useful for producing flexible markup but do not justify the definition of queries to challenge them directly. Therefore, we only made use of a restricted set of XML features in the data generator which we consider performance critical in the context of XML processing in databases. We do not generate documents with Entities or Notations. Neither do we distinguish between Parsed Character Data and Character Data assuming that both are string types from the viewpoint of the storage engine. Furthermore, we don't include namespaces into the queries. We also restrict ourselves to the seven bit ASCII character set. A DTD and schema information are provided to allow for more efficient mappings. However, we stress that this is additional information that *may* be exploited.

4.5 `xmlgen` – Or How to Generate a Document

We designed and implemented a document generator, called `xmlgen`, to provide for a scalable XML document database. Besides the obvious requirement to be capable of producing the XML document specified above we were eager to meet the following additional demands. The generation of the XML document should be: (1) *platform independent* so that any user interested in running the benchmark is able to download the binary and generate the same document no matter what hardware or operating system is used; to achieve this plain ANSI C was used to implement `xmlgen`; (2) *accurately scalable* ranging from a minimal document to any arbitrary size limited only by the capacity of the system; (3) both *time and resource efficient*,

Name	Scaling Factor	Document Size
tiny	0.1	10 MB
standard	1	100 MB
large	10	1 GB
huge	100	10 GB

Figure 3: Scaling the benchmark document

i.e., elapsed time ideally scales linearly whereas the resource allocation is constant – independent of the size of the generated document; (4) *deterministic*, that is, the output should only depend on the input parameters.

First, in order to be able to reproduce the document independently of the platform, we incorporated a random number generator rather then relying on the operating system's built-in random number generators. Together with basic algorithms which can be found in statistics textbooks this `xmlgen` implements uniform, exponential, and normal distributions of fairly high quality. We assigned to each of the elements in the DTD a plausible distribution of its children and its references, observing consistency among referencing elements, that is, the number of items organized by continents equals the sum of open and closed auctions, *etc*. Second, to provide for accurate scaling we scale selected sets like the number of items and persons with the user defined factor. Moreover, we calibrated the numbers to match a total document size of slightly more than 100 MB for scaling factor 1.0 (*cf.* Fig. 3). Finally, it is a challenge to implement the data generator efficiently because references are created at various places throughout the document; since we have to abide by the integrity constraint that every reference points to a valid identifier we could go for the straight-forward solution of keeping some sort of log and record which identifier has already been referenced; unfortunately this seems infeasible for large documents. We solved this problem by modifying the random number generation to produce several identical streams of random numbers. That way, we are able to implement a partitioning of sets like the item IDs that are referenced from both open and closed auctions. In its current version, `xmlgen` requires less than 2 MB of main-memory, and produces documents of sizes of 100 MB and 1 GB in 33.4 and 335.5 seconds, respectively (450MHz Pentium III). A more detailed description of the tool and downloads can be found at the project Web page [18].

5 Bulkloading the Document

In the context of XML, the role of bulkloading stands apart from its importance in other benchmarks. As (prospective) standards like XQuery [11] are not exclusively designed as database but data integration languages, we can, strictly speaking, neither assume the need to bulkload documents nor the presence of a database. However, there should be little doubt that databases can help with managing large amounts of XML data. We therefore resort to the following interpretation: We take the XQuery syntax for

```
FOR $a in document("auction.xml")/...
```

literally and formulate all queries with respect to a single large document without committing ourselves to a specific database scenario.

Although the authors themselves did not experience problems when bulkloading the document in our test environments, they are aware that the size of the document may be too large for some systems. Hence, the data generator `xmlgen` additionally offers a mode that outputs n entities (as defined in Section 4) per file where n can be chosen by the user.

Note that in this case, modifications to the one-document version of the benchmark may become necessary. For example, if the user chooses to make use of the DTD we supply, parser-controlled references, *i.e.*, ID and IDREF declared attributes, should be converted to REQUIRED attributes. Otherwise a validating parser tries to check for uniqueness and existence of IDs and IDREFs, respectively. With respect to queries, changes to the path expressions, which as presented assume a single document, are necessary. Nevertheless, all changes remain local. However, we stress that this solution should be regarded as a work-around and that the semantics of the queries as defined in the following section should not differ no matter whether they are executed against a single document or a collection of documents. The query semantics of the *one* document version are normative.

6 Benchmark Queries

This section lists the queries of the benchmark. We chose to express the queries in XQuery [11], the successor to Quilt [12], which is about the be standardized. Due to lack of space, this is the only query we present in source code. The remaining queries can be downloaded from the project Web site at [19]. The Queries are grouped under subsection headings which indicate the concept to be tested.

6.1 Exact Match

This simple query is mainly used to establish a performance baseline, which should help to interpret subsequent queries. It tests the database ability to handle simple string lookups with a fully specified path.

Q 1. *Return the name of the person with ID 'person0'.*

```
FOR      $b IN /site/people/person/[@id="person0"]
RETURN $b/name/text()
```

Now we are ready for more challenging queries:

6.2 Ordered Access

These queries should help users to gain insight how the DBMS copes with the intrinsic order of XML documents and how efficiently they can expect the DBMS to handle queries with order constraints.

Q 2. *Return the initial increases of all open auctions.*

This query evaluates the cost of array lookups. Note that it may actually be harder to evaluate than it looks; especially relational back-ends may have to struggle with rather complex aggregations to select the bidder element with index 1.

Q 3. *Return the first and current increases of all open auctions whose current increase is at least twice as high as the initial increase.*

This is a more complex application of array lookups. In the case of a relational DBMS, the query can take advantage of set-valued aggregates on the index attribute to accelerate the execution. Queries Q2 and Q3 are akin to aggregations in the TPCD [15] benchmark.

Q 4. *List the reserves of those open auctions where a certain person issued a bid before another person.*

This time, we stress the textual nature of XML documents by querying the tag order in the source document.

6.3 Casting

Strings are the generic data type in XML documents. Queries that interpret strings will often need to cast strings to another data type that carries more semantics. This query challenges the DBMS in terms of the casting primitives it provides. Especially, if there is no additional schema information or just a DTD at hand, casts are likely to occur frequently. Although other queries include casts, too, this query is meant to challenge casting in isolation.

Q 5. *How many sold items cost more than 40?*

6.4 Regular Path Expressions

Regular path expressions are a fundamental building block of virtually every query language for XML or semi-structured data. These queries investigate how well the query processor can optimize path expressions and prune traversals of irrelevant parts of the tree.

Q 6. *How many items are listed on all continents?*

A good evaluation engine or path encoding scheme should help realize that there is no need to traverse the complete document tree to evaluate such expressions.

Q 7. *How many pieces of prose are in our database?*

Also note that `COUNT` aggregations do not require a complete traversal of the document tree. Just the cardinality of the respective parts is queried.

6.5 Chasing References

References are an integral part of XML as they allow richer relationships than just hierarchical element structures. These queries define horizontal traversals with increasing complexity. A good query optimizer should take advantage of the cardinalities of the operands to be joined.

Q 8. *List the names of persons and the number of items they bought. (joins person, closed_auction)*

Q 9. *List the names of persons and the names of the items they bought in Europe. (joins person, closed_auction, item)*

6.6 Construction of Complex Results

Constructing new elements may put the storage engine under stress especially when the newly constructed elements are to be queried again. The following query reverses the structure of person records by grouping them according to the interest profile of a person. Large parts of the person records are repeatedly reconstructed. To avoid simple copying of the original database we translate the mark-up into French.

Q 10. *List all persons according to their interest; use French markup in the result.*

6.7 Joins on Values

This query tests the database's ability to handle large (intermediate) results. This time, joins are on the basis of values. The difference between these queries and the reference chasing queries Q8 and Q9 is that references are specified in the DTD and may be optimized with logical OIDs for example. The two queries Q11 and Q12 differ mainly in the size of the result set and hence provide various optimization opportunities.

Q 11. *For each person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.*

Q 12. *For each person with an income of more than 50000, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.*

6.8 Reconstruction

A key design for XML-to-DBMS mappings is to determine the fragmentation criteria. The complementary action is to reconstruct the original document from its broken-down representation. Query 13 tests for the ability of the database to reconstruct portions of the original XML document.

Q 13. *List the names of items registered in Australia along with their descriptions.*

6.9 Full Text

We continue to challenge the textual nature of XML documents; this time, we conduct a full-text search in the form of keyword search. Although full-text scanning could be studied in isolation we think that the interaction with structural mark-up is essential as the concepts are considered orthogonal; so Q14 is restricted to a subset of the document by combining content and structure.

Q 14. *Return the names of all items whose description contains the word 'gold'.*

6.10 Path Traversals

In contrast to Section 6.4 we now try to quantify the costs of long path traversals that don't include wildcards. We first descend deep into the document tree (Query 15); in Query 16 we additionally ascend the tree with a selection. Note that both queries only check for the existence of paths.

Q 15. *Print the keywords in emphasis in annotations of closed auctions.*

Q 16. *(Confer Q 15.) Return the IDs of the sellers of those auctions that have one or more keywords in emphasis.*

6.11 Missing Elements

This is to test how well the query processors know to deal with the semi-structured aspect of XML data, especially elements that are declared optional in the DTD.

Q 17. *Which persons don't have a homepage?*

The fraction of people without a homepage is rather high so that this query also presents a challenging path traversal to non-clustering systems.

6.12 Function Application

This query puts the application of user defined functions (UDF) to the proof. In the XML world, UDFs are of particular importance because they allow the user to assign semantics to generic strings that go beyond type coercion.

Q 18. *Convert the currency of the reserves of all open auctions to another currency.*

6.13 Sorting

Due to the lack of a schema, SORTBY clauses often play the role of the SQL-ish ORDER BY and GROUP BY. This query requires a sort on generic strings.

Q 19. *Give an alphabetically ordered list of all items along with their location.*

6.14 Aggregation

The following query computes a simple aggregation by assigning each person to a category. Note that the aggregation is truly semi-structured as it also includes those persons for whom the relevant data is not available.

Q 20. *Group customers by their income and output the cardinality of each group.*

These twenty queries constitute the challenges posed in XMark. While deploying them in various environments we felt that the number and type of queries exhibit a good balance between conciseness and detail, making it possible to run the benchmark in acceptable time while still acquiring interesting characteristics of the system(s) tested.

7 Experiments and Experiences

The benchmark has been a group-design activity of academic and industry researchers and is known to be used with success to evaluate progress in both commercial and research settings. The evaluation in this section here is meant to present the highlights we encountered when running the benchmark on a broad range of the systems; an in-depth analysis of the behavior of all individual systems would be beyond the scope of this paper. We anonymized the systems due to well-known license restrictions; instead, we simply speak of systems A through G. Our test platforms fall into two categories: (1) Systems that are designed as *large scale repositories* and therefore can be expected perform well at handling large amounts of data. We call them System A through System F; in the sequel, we will also refer to these systems as *mass storage systems.* Some of the systems, namely A to C, are based on relational technology, come with a cost-based query optimizer and allow the kind of hand-optimization and hints as the relational product. While A and B do not require the user to provide a mapping for physical data breakdown, System C reads in a DTD and lets the user generate an optimized database schema. Systems D to F are main-memory based and only come with heuristic optimizers; however, they also allow rewriting the queries by hand if necessary. (2) Query processors that are intended to serve as *embedded query processors* in programming languages and aim at small to medium sized documents. We call the software system that falls into this category System G.

A note on the analysis. Some systems provided us with the opportunity to look at query execution in detail, *i.e.*, find out how much time is spent for query optimization, metadata access or during I/O wait; others only allowed a black-box analysis augmented with the usual monitoring tools that operating systems provide. The tools to run the benchmark document have been made available on the project Web site [18]. They include the data generator and the query set along with a mapping tool to convert the benchmark document into a flat file that may be bulk-loaded into a (relational) DBMS; a variety of formats are available.

The experiments we conducted are based on a variety of set-ups: some systems required us to prepare the data and translate the queries into a proprietary language that was then executed, other systems processed the queries as they are presented in [19, 22] possibly with minor syntactic changes. All queries were run on machines equipped with 550 MHz Pentium III processors, SCSI Ultra2 harddisks and 1 GB of main memory; operating systems were Windows 2000 Advanced Server and Linux 2.4 respectively depending on what the packages required. Although the systems were all equipped with at least two processors, only one processor was used during bulk load and query execution.

System	Size	Bulkload time
A	241 MB	414 s
B	280 MB	781 s
C	238 MB	548 s
D	142 MB	50 s
E	302 MB	96 s
F	345 MB	215 s

Table 1: Database sizes

Concerning the scaling factor, we were not able to run all queries on all systems at scaling factor 1.0 as we had intended to do. The mass storage Systems A-F were able to process the queries, but the embedded System G failed to do so. So for the Systems A-F, which with we commence our evaluation, the benchmark document was generated at scaling factor 1.0. Note that it took the XML parser `expat` [13] 4.9 seconds (user time on the above Linux machine including system time and disk I/O) to scan the benchmark document (this time only includes the tokenization of the input stream and normalizations and substitutions as required by the XML standard and no user-specified semantic actions). The bulkload times are summarized in Table 1: they range from 49 seconds to 781 seconds. They constitute completed transactions and include the conversion effort needed to map the XML document to a database instance. Note that System C requires a DTD to derive a database schema; the time for this derivation is not included in the figure, but is negligible anyway. The resulting database sizes are also listed in Table 1; we remark that some systems which are not included in this comparison require far larger database sizes.

We now turn our attention to the running times and statistics as displayed in Table 3 and present some basic insights. Since we do not have the space to discuss all timings and experiments in detail, we only present

Query	System	Compilation CPU	Compilation total	Execution CPU	Execution total
Q 1	A	16%	25%	31%	75%
	B	13%	51%	30%	49%
	C	0%	29%	20%	71%
Q 2	A	9%	13%	41%	87%
	B	12%	20%	65%	80%
	C	3%	16%	77%	84%

Table 2: Detailed timings of Q1 and Q2 for Systems A, B, C

	System A	System B	System C	System D	System E	System F
Q 1	689	784	257	120	1597	2814
Q 2	3171	1971	707	2900	4659	7481
Q 3	41030	6389	1942	3900	4630	8074
Q 5	259	221	237	160	246	204
Q 6	293	331	509	10	336	508
Q 7	719	741	1520	10	287	2845
Q 8	1684	1466	667	470	3849	9143
Q 9	3530	10189	92534	980	5994	13698
Q 10	3414285	86886	1568	22000	54721	69422
Q 11	205675	2551760	2533738	8700	602223	741730
Q 12	126127	965118	976026	7500	268644	270577
Q 17	1008	1117	240	250	2103	3598
Q 20	821	939	1254	620	1065	1759

Table 3: Performance in ms of queries discussed in Section 7

a selection. In most physical XML mappings found in the literature, Query Q1 consists of a table scan or index lookup and a small number of additional table look-ups. It is mainly supposed to establish a performance baseline: At scaling factor 1.0, the scan goes over 10000 tuples and is followed by two table lookups if a mapping like [20] is used.

Queries Q2 and Q3 are the first ones to provide surprises. It turns out that the parts of the query plans that compute the indices are quite complex TPC/H-like aggregations: they require the computation of set-valued attributes to determine the bidder element with the least index with respect to the open auction ancestor. Therefore the complexity of the query plan is higher than the rather innocent looking XQuery representations of the queries might suggest. Consequently, running times are quite high. Although System A was able to find an execution plan which was as good as that of the other systems, it spent too much of its time on optimization. Table 2 displays some interesting characteristics of Q1 and Q2 that can be traced back to the physical mappings the systems use. System A basically stores all XML data on one big heap, *i.e.*, only a single relation. System B on the other hand uses a highly fragmenting mapping. Consequently, System A has to access fewer metadata to compile a query than System B, thus spending only half as much time on query compilation (*i.e.*, optimization) as System B. However, this comes at a cost. Because the data mapping deployed in System A has less explicit semantics, the actual cost of accessing the real data is higher than in System B (75% *vs* 49%). System C as mentioned needs a DTD to derive a storage schema; this additional information helps to get favorable performance. Still in Table 2, we also find the detailed execution times for Q2. They show that mappings that structure the data according to their semantics can achieve significantly higher CPU usage (compare 77% of System C and 65% of System B *vs* System A's 41%). We remark that System C also uses a data mapping in the spirit of [23] that results in comparatively simple and efficient execution plans and thus outperforms all other systems for Q2 and Q3.

Q4 (no results presented due to lack of space) features the 'BEFORE' predicate which may be expensive to evaluate. Some mappings like [26] which store the *extent* of tags, *i.e.*, not only the position of the start tag but also that of the corresponding end tag, may be able to exploit this additional information and achieve good running times.

We now come to Query Q5 which tries to quantify the cost of casting or type-coercion operations such as those necessary for the comparisons in Q3. For all mass-storage systems, the cost of this coercion is rather low with respect to the relative complexity of Q3's query execution plan and given the execution times of Q5. In any case, Q5 does not exhibit great differences in execution times. We note that all character data in the original document, including references, were

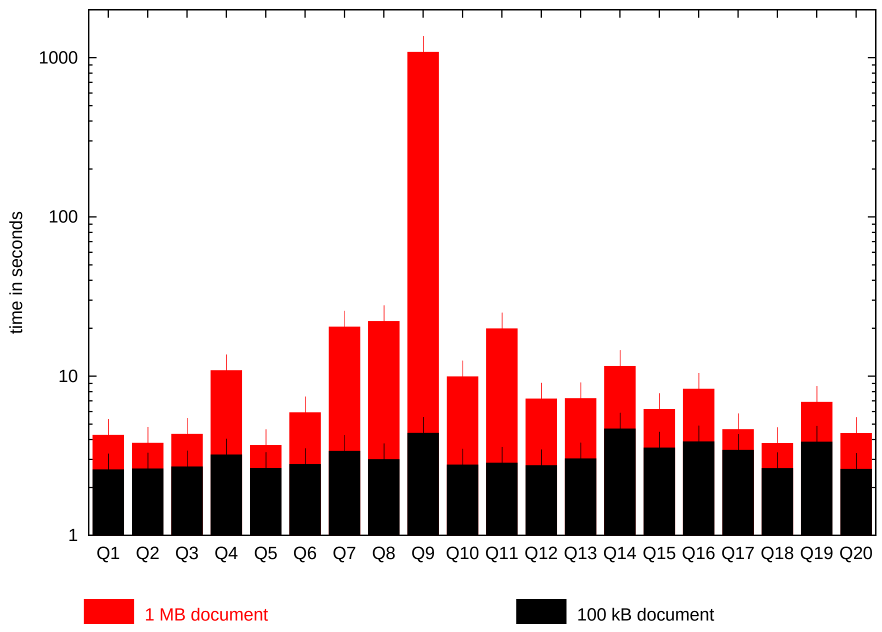

Figure 4: Performance figures for the embedded query processor System G

stored as strings and cast at runtime to richer data types whenever necessary as in Queries 3, 5, 11, 12, 18, 20. We did not apply any domain-specific knowledge; neither did the systems use schema information nor precalculation or caching of casting results.

Regular path expressions are the challenge presented by queries Q6 and Q7. System D keeps a detailed structural summary of the database and can exploit it to optimize traversal-intensive queries; this actually makes Q6 and Q7 surprisingly fast. However, on systems without access to structural summaries, which effectively play the role of an index or schema, these queries often are significantly more expensive to execute. The problem that Q7 actually looks for non-existing paths is efficiently solved by exploiting the structural summary in the case of System D. For some systems, the cost of accessing schema information was very high and dominated query performance.

Queries Q8 and Q9 are usually implemented as joins. In the systems that we could analyze in detail, chasing the references basically amounted to executing equi-joins on strings. We were surprised that Q8 and Q9 were relatively cheap in comparison to Q2 and Q3 as we would have deemed the individual elements similarly expensive. For Q9, System C was not able to find a good execution plan in acceptable time. Apart from that anomaly, the implementation of the executed join algorithms seemed to determine the performance.

The construction of complex query results is addressed in Q10. The path expressions and join expression used in the query are kept simple so that the bulk of the work lies in the construction of the answer set which amount to more than 10 MB of (unindented) XML text.

Whereas Q10 produced massive amounts of output data, Q11 and Q12 test the ability to cope with large intermediate results by theta-joining potential buyers and items that might be of interest to them. The theta-join produces more than 12 million tuples. Q12 especially is also a challenge to the query optimizer to pick a good execution plan and allows insights into how the data volume influences query and output performance. For Systems B and C, the optimizer chose a sub-optimal execution plan. For Systems D through F we had to experiment with several hand-optimized execution plans.

The queries so far have not attempted to estimate the cost of long path expressions with respect to short ones. This is exactly where Q15 and Q16 can help. For the sake of space, we do not present figures here but report that Systems A, B and C needed about 8 times longer to execute Q16 than they needed for Q15. This is due to the many joins that the more complicated path expression in Q16 brings about – both in execution and optimization. Also, the SQL output of System E needed some massaging to keep intermediate results small.

Q17 again stresses the loose schema of many XML documents by querying for non-existing data. The query execution plan computes the intersection of two sets. The timings in Table 3 show a typical situation: although all systems are able to process the query in less than four seconds, there is still an order magnitude of difference in the performance. Queries Q18 and Q19 (no performance figures are presented due to lack of space) are primarily of interest to establish the relative costs of function application and sorting operations when comparing two system architectures. The aggregations of Q20 conclude the query set with a combination of three table scans and a set difference. All systems show similar performance.

A general note on the queries: we often had to reformulate the queries into SQL or proprietary XML query languages to satisfy a query processor; frequently it was also necessary to hand-optimize multi-pass SQL, *i.e.*, sequences of SQL statements that reuse intermediate results, that were generated by the 'native' XML engines E and F. Systems A to C on the other hand did not need manual intervention. As a separate observation, we would like to mention that the installation effort for the tested systems differed greatly and that in a production setting this effort may be very important [21].

In some of the performance figures certain systems (particularly Systems A to C) show pathological running times. This does not necessarily mean that the relevant systems are inferior to the others; we rather relied on the built-in query optimizers and did not at all change or reformulate queries by hand. This was to show that the benchmark queries indeed present reasonable challenges that can be solved even if not optimally. The analysis of the query translation and optimization process showed that search spaces for XML queries are often larger than necessary since, during the translation from XQuery to a lower-level algebra, information especially about path expression is often lost. To improve on this, experimenting with new pruning strategies and extended low-level algebras to better capture query semantics, might be a good starting point.

For comparison, Figure 4 presents the performance behavior of an embedded query processor on document sizes 100 kB (scaling factor 0.001) and 1MB (scaling factor 0.01), which were the largest sizes we could sensibly execute on the System G without running out of resources. On the smaller document, no query took longer than 5 seconds but none was faster than 2.5 seconds; this means that the implementation techniques used for the embedded processor incur a significant performance overhead compared to the mass storage systems, which were overall competitive in many cases. With respect to join queries one should note that the result sizes were very small in comparison to those of the mass storages systems, on which we used a much larger scaling factor (1.0) for our experiments. An advantage of the embedded systems is that they usually allow more control over query execution by providing hooks into their execution engine or by letting users implement certain operators themselves.

We also would like to mention one point that would facilitate query formulation enormously. If a query processor was able to validate path expressions online, *i.e.*, tell the user whether a given sequence of tags actually exists in the database instance, it would often be of great help to users as quite regularly, simple typos in path names often evaluate to empty results. While the DBMS of course can't decide whether a given path expression contains a typo or not, it could well issue a warning if a path expression contains non-existing tags. An approach like Query By Example could possibly lead to very helpful results.

8 Conclusion

In this paper, we presented the benchmark specifications developed for XMark, a set of queries and lessons learned while executing the workload specification on a number of platforms. The benchmark was designed top-down taking the standardization issues around XML as a starting point. The queries try to capture the essential primitives of XML processing in Data Management Systems such as path expressions, querying for NULL values, full-text search, hierarchical data with varying fan-outs, ordered data and coercions between data types and complex results.

Our experiences during the experiments can be summarized as follows: (1) The physical XML mapping has a far-reaching influence on the complexity of query plans. Each mapping favors certain types of queries by enabling efficient execution plans for them. However, no mapping was able to outperform the others across the board. (2) The complexity of query plans is often aggravated by information-loss during translation from the declarative high-level query language to the low-level execution algebra. There often appears to be a semantic gap between the two. Thus, cost-based query optimizers have to deal with larger search spaces than necessary. (3) Meta-data access can be a dominant factor in query execution especially in simple lookup queries with small result sizes (see Table 2 and explanations in Section 7). (4) Schema information often enables better database schema design and is also useful in query optimization; see the running times of System C versus the comparable Systems A and B in Table 3.

Important parts of a complete application scenario are still missing: update specifications, for which a W3C standard has yet to be defined, are the most prominent one. Therefore, we expect our work to continue and evolve in the future along with the standardization efforts.

References

[1] T. Anderson, A. Berre, M. Mallison, H. Porter, and B. Schneider. The HyperModel Benchmark. In *International Conference on Extending Database Technology*, volume 416 of *Lecture Notes in Computer Science*, pages 317–331, 1990.

[2] T. Böhme and E. Rahm. XMach-1: A Benchmark for XML Data Management. In *Proceedings of BTW2001*, 2001.

[3] A. Bonifati and S. Ceri. Comparative Analysis of Five XML Query Languages. *ACM SIGMOD Record*, 29(1):68–79, 2000.

[4] R. Bourett. XML Database Products. available at `http://www.rpbourret.com/xml/XMLDatabaseProds.htm`, 2000.

[5] J. Boyer. *Canonical XML Version 1.0*, 1 2001. available at `http://www.w3.org/TR/xml-c14n`.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). available at `http://www.w3.org/TR/REC-xml`, 2000.

[7] S. Bressan, G. Dobbie, Z. Lacroix, M. Lee, Y. Li, and U. Nambiar. X007: Applying 007 Benchmark to XML Query Processing Tools. In *International Conference on Information and Knowledge Management*, pages 167–174, 2001.

[8] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 12–21, 1993.

[9] M. Carey, D. DeWitt, J. Naughton, M. Asgarian, P. Brown, J. Gehrke, and D. Shah. The BUCKY Object-Relational Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 135–146, 1997.

[10] R. Cattell and J. Skeen. Object Operations Benchmark. *TODS*, 17(1):1–31, 1992.

[11] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery: A Query Language for XML, February 2001. available at `http://www.w3.org/TR/xquery`.

[12] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *International Workshop on the Web and Databases (WebDB)*, pages 53–62, 2000.

[13] James Clark et al. Expat XML Parser. available at `http://sourceforge.net/projects/expat/`, 2001.

[14] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[15] J. Gray. Database and Transaction Processing Performance Handbook. available at `http://www.benchmarkresources.com/handbook/contents.asp`, 1993.

[16] M. Klettke and H. Meyer. XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. In *International Workshop on the Web and Databases (WebDB)*, pages 63–68, 2000.

[17] K. Ramasamy, J. Patel, J. Naughton, and R. Kaushik. Set Containment Joins: The Good, The Bad and The Ugly. In *Proceedings of the International Conference on Very Large Data Bases*, pages 351–362, 2000.

[18] A. Schmidt, M. Kersten, D. Florescu, M. Carey, I. Manolescu, and F. Waas. The XML Store Benchmark Project, 2000. `http://www.xml-benchmark.org`.

[19] A. Schmidt, M. Kersten, D. Florescu, M. Carey, I. Manolescu, and F. Waas. Example Snippet and Queries, 2002. available at `http://monetdb.cwi.nl/xml/snippet.txt` and `http://monetdb.cwi.nl/xml/queries.txt`.

[20] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient Relational Storage and Retrieval of XML Documents. In *International Workshop on the Web and Databases (WebDB)*, pages 47–52, Dallas, TX, USA, 2000.

[21] A. Schmidt, F. Waas, M. Kersten, D. Florescu, M. Carey, I. Manolescu, and R. Busse. Why And How To Benchmark XML Databases. *ACM SIGMOD Record*, 30(3):27–32, 2001.

[22] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, April 2001.

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

[24] D. R. Slutz. Massive Stochastic Testing of SQL. In *Proceedings of the International Conference on Very Large Data Bases*, pages 618–622, 1998.

[25] W3C. W3C XML Schema. `http://www.w3.org/XML/Schema`, 2001.

[26] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.