A Look Back on the XML Benchmark Project

Albrecht Schmidt[1], Florian Waas[2], Stefan Manegold[3], and Martin Kersten[3]

[1] Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst
al@cs.auc.dk
[2] Microsoft Corporation
Redmond (WA), USA
florianw@microsoft.com
[3] Centre for Mathematics and Computer Science (CWI)
Kruislaan 413, NL-1098 SJ Amsterdam
firstname.lastname@cwi.nl

1.1 Introduction

Database vendors and researchers have been responding to the establishing of XML [6] as the premier data interchange language for Internet applications with the integration of XML processing capabilities into Database Management Systems. The new features fall into two categories: *XML-enabled interfaces* allow the DBMS to speak and understand XML formats, whereas *XML extensions* add novel primitives to the engine core. Both kinds of innovations have the potential to impact the architecture of software systems, namely by bringing about a complexity reduction in multi-tier systems. However, it is often difficult to estimate the effect of these innovations. This is where the XML Benchmark Project tries to help with XMark. By providing an application scenario and a query workload, the benchmark suite can be used to identify strengths and weaknesses of XML-enabled software systems.

The queries of the benchmark suite target different aspects of querying of XML documents, both in isolation and in combination. We identify the following areas of potential performance impacts:

– The topology of XML structures as found in the original document is a potential candidate for queries; especially systems that implement document order on top of an unordered data model may not be properly prepared for this kind of challenge and have to turn rather simple queries into complex operations. This is also tested in several benchmark queries.
– The document-oriented nature of XML makes strings the basic data type applications have to deal with. Typing XML documents is therefore as important a challenge to make data processing more robust as enforcing other semantics constraints. Problems can also arise as the typing rules of query languages may clash with the more complex type systems of host programming languages. In

addition, strings are often not efficient in database systems since their length can vary greatly, putting additional stress on the storage engine.
- The hierarchical structure of XML documents impacts queries in the form of path expressions. The hierarchical structure of documents in conjunction with complicated path expressions does not only result in potentially expensive join and aggregation operations but also in a search space that makes it hard for query optimizers to find good execution plans.
- The loose schema of XML data may not only make it hard for users to get an overview of the structure, which is a prerequisite for being able to user query languages sensibly and a notoriously error-prone activity when a user tries to specify long and complicated path expressions. It also poses optimization challenges to the database engine. Sparsely populated parts of the database do not only aggravate maintenance problems with respect to data statistics, they also inflate the size of the database with maintenance information and NULL values.
- Besides the tree structure, XML provides a number of additional features that influence a query processor. For example, the XML standard lists constraints on special attributes to ensure that references only connect existing elements. To cope with references efficiently, techniques like join indexes or logical OIDs might be of use. The resolution of namespaces is another topic that requires careful handling; XMark does not feature queries that challenge namespaces since its authors believe that the mechanism to handle them do not differ greatly from queries involving different parts of subtrees.

Due to complex interdependencies between these points and the different components of a system, implementation efforts tend to be hard to assess in a general fashion without putting them to a standardized test, which is most conveniently done in the form of a benchmark. The need for new benchmarks has been a recurring momentum in database research; consequently, over the past years the database community developed a rich tradition in performance assessment of systems ranging from research developments like the Hypermodel [2], 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 fail to take into account the flexibility and expressiveness of semi-structured data with their implicit schemas [1] and flexible data structures which exceed the capabilities of existing query languages.

The XMark Benchmark 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 underlying algebra. 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 or transformations to the output. As for the choice of language, we use XQuery [12] which is the result of incorporating experiences from various research languages [4] for semi-structured data and XML into a standard.

The target audience of the benchmark could comprise 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.

1.2 Evolution of XML Technology and Benchmarks

Database benchmarks found in the literature cover a plethora of technologies and 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. Therefore, to motivate the need for XML benchmarks we take a look at the evolution of XML standards and how it differs from that of established technologies.

The evolution of XML differs significantly from the evolution of relational databases in that for XML there was an early standard which was accepted and supported by a large community. It was then that implementations had to live up to the standards that were already present and in place. There was no organic and interactive development between standards and research as there was, for example, in the case of the SQL standards. Therefore it is sensible to design the benchmark with a top-down perspective in mind, *i.e.*, to come up with challenges for query primitives anticipated as typical and thus provide a kind of thematic benchmark. In the case of XML, the W3C Use Cases [11] contained the research necessary to justify the relevance of the challenges. In this sense, the combination of traditional and new features present in XML processing systems in conjunction with the new approach to standards results in the need for a new quality of system development. The XMark benchmark tries to be a part of this endeavor.

Traditionally, database management systems have been deployed in settings where very regular, table-structured data format prevail. While it has been shown that these data-centric documents, *i.e.*, documents which logically represent data structures [5], map effectively 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 more document-centric [5], *i.e.*, consisting mostly of natural language with mark-up only interspersed with the result of irregular path structures. Converted to relational tables in a naive way, the data and query profile often do not match the kind of pattern traditional database engines are optimized for.

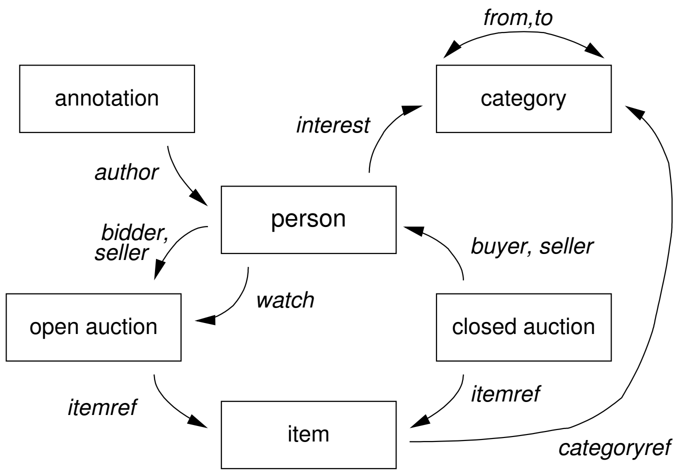

Fig. 1.1. Overview of main entities in the XMark document

1.3 The XMark Database

In this section, we give an overview of XMark's document database. One of the major design goals were good performance during generation, opportunities for formulating predictable and interesting queries, and that the database 'feels natural'. We first summarize the semantics of the document and then cover some of the more technical issues of generating such documents.

1.3.1 Main Entities and Their Relationships

The main syntactic constituent of XML documents is the recursive application of elements that contain other elements; it renders the typical tree structure of XML. Accordingly, element relationships play a crucial role in document design. The other important tool that designers have at hand are references which connect elements in a way that is orthogonal to the tree structure. In XMark, we decided to model the database after a schema that is typical for Internet auction sites. The main entities come in two groups: *person*, *open auction*, *closed auction*, *item*, and *category* on the one side and entities akin to *annotation* on the other side. The relationships between the entities in the first group are expressed through references, as depicted with arrows in Figure 1.1. The relationships between the entities 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. An ER diagram can be found in [7]. The entities we just mentioned carry the following semantics:

– *Items* are the objects that are on sale in an auction 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 element.

– *Open auctions* are auctions in progress. Their properties are the privacy status, the bid history (*i.e.*, increases or decreases over time) 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.
– *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.
– *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.
– *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.

We emphasize that these entities constitute the relatively structured, *i.e.*, data-oriented part of the document. Their sub-element structure is fairly regular on a per entity basis but there are predictable exceptions such as that not every person has a homepage; in a relational DBMS, these exceptions would typically be taken care of by NULL values. Another characteristic of these entities is that, apart from occasional list types such as bidding histories, the order of the input is not particularly relevant. On the other hand, the sub-elements of the document-centric part of the database, namely those of *annotation* and similar elements, do not accentuate the above aspects. Here the length of strings and the internal structure of sub-elements varies greatly. The markup now comprises itemized lists, keywords, and even visual formatting instructions and character data, doing its best to imitate the characteristics of natural language texts. This warrants that the benchmark database covers the full range of XML instance incarnations, from marked-up data structures to traditional prose.

The arrows in Figure 1.1 are mainly implemented as IDREFs that connects elements with IDs. 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 although this constraint does not materialize in the DTD that accompanies XMark.

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.

1.3.2 The Document Generator

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:

– *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`;
– *accurately scalable* ranging from a minimal document to any arbitrary size limited only by the capacity of the system;
– both *time and resource efficient*, *i.e.*, elapsed time ideally scales linearly whereas the resource allocation is constant – independent of the size of the generated document;
– *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 the data generator `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 at scaling factor 1.0. 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 the 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 on the project Web page [18].

1.4 The XMark Queries

In total, XMark contains 20 queries testing various concepts such as exact match, ordered access, casting, regular path expressions, chasing references, construction of complex results, joins on values, reconstruction, full text search, path traversals, missing elements, function application, sorting, and aggregation. Due to lack of space, we present only a representative selection of the queries, here. A complete description of all queries is available in [21] and the respective XQuery-code can be downloaded from the project Web site at [19].

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.

Q1: *Return the name of the person with ID 'person0'.*

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.

Q2: *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.

Q3: *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.

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.

Q5: *How many sold items cost more than 40?*

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.

Q6: *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.

Q7: *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.

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.

Q8: *List the names of persons and the number of items they bought. (joins person, closed_auction)*

Q9: *List the names of persons and the names of the items they bought in Europe. (joins person, closed_auction, item)*

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.

Q10: *List all persons according to their interest; use French markup in the result.*

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.

Q11: *For each person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.*

Q12: *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.*

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.

Q17: *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.

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.

Q20: *Group customers by their income and output the cardinality of each group.*

1.5 Experiences and Lessons Learned

In this section, we summarize some of the experiences we gathered during the design of the benchmark and when we ran the setup on a number of platforms.

1.5.1 Benchmark Document

In past database benchmarks, there have been two main routes to designing a database. On the one hand, designers may lean towards databases that exhibit properties close to what is found in real-world applications. This has the advantage that queries feel natural and that it is hard to question the usefulness of the scenario. On the other hand, it is often desirable to have another property which often is hard to combine with naturalness, namely predictable query behavior. If designers pursue predictability, they often go for very regular designs so that they can exactly and reliably predict what queries return. These designs are frequently based on mathematical models which allow precise predictions – at times at the trade-off however that the resulting databases ‘feel’ only little natural.

It is hard to position XML between the two extremes. For one, XML is not a pure machine format and therefore not exclusively consumed and produced by applications but also absorbed by humans – at least occasionally. Therefore, not only the semantics but also the documents themselves should still make sense to humans while it is primarily machines that produce and consume them. In XMark, we thus tried to reconcile the two competing goals as much as possible but, in case of conflicts, our policy was to favor predictability of queries and performance in the generation process.

We should mention that designers of other XML benchmarks had different policies in mind. For example, the Michigan Benchmark [17] features a very structured approach to database generation and want to maximize predictability on all levels and queries, much in the spirit of the Wisconsin Benchmark described in [15]. A hybrid approach is taken by XBench [24] who classify their documents according to a requirements matrix: their axes are Single-Document *vs*. Multi-Document Databases and Text-Centric *vs*. Data-Centric Databases, respectively. Other XML benchmarks like X007 [7] and XMach-1 [3] are also based on certain considerations with respect to document design.

While most people agree that performance is an important goal in query execution, it is equally important in data generation especially when it comes to large databases, which bring about significant generation overhead. In XMark, we pursued performance in that it was a design goal that the data generator should be able to output several megabytes of XML text per second, which we considered a necessary requirement should it be suitable for deployment in large-scale scenarios. After we finished a first prototype of the generator, we found out that a major performance bottleneck was random number generation. At first, we had chosen a high quality random number generator which turned out to be inadequate. In the sequel, we had to deal with the trade-off between the quality of random variables in general and their correlations in particular at one end of the scale and generation time at the other end. What turned out to be a problem was that when weak correlations were to be generated the quality of the random number generator may not be sufficient to make a correlation actually materialize in the generated database instance. On the other hand, using high-quality random number generators may be too costly in terms of resource consumption. We tackled the issue by fine-tuning the parameters that define the available scaling factors. As a final remark on correlations, we mention that it is quite easy to specify correlations between different entities that are logically sound but nearly impossible to materialize in the generated database due to the above-mentioned constraints. Hence, it is important to find a logic to describe or model the benchmark database that at the same time is non-contradictory and feasible. In the design of `xmlgen`, XMark's data generator, we put considerable effort into finding both economical and reliable ways of generating random numbers. Especially, `xmlgen` makes use of reproducible streams of random numbers to ensure that important correlations are preserved, most notably the well-formedness constraints imposed by the XML standard; technically,, we use deterministic number generation. When a fully customizable document generator is used, the language describing the document may contain contradictions; in this case, an important design rationale is to eliminate or minimize the contradictions, for example, by reporting them to the user through warning and error messages. We believe that there are still many open research issues with respect to data generation. Promising subjects include how to generate interesting chain correlations in large data sets and statistical guarantees for their materialization in the data sets, amongst others.

1.5.2 Running the Benchmark

Since XML was still at an early stage in its development, the actual implementation of the benchmark on a number of systems was a non-trivial task. The architectures and capabilities of query processors very much varied from system to system. Some systems could only bulkload small documents at a time; hence, we sometimes had to use the split feature of the data generator and feed the benchmark document in small pieces; at other times we were given the opportunity to specify (parts of) the XML-to-Database mapping by hand. The benchmark queries (see [22] for a complete list) often had to be translated to standard (SQL and XQuery) or proprietary query languages and possibly annotated with execution plan hints. All in all, there were many opportunities for hand-optimization which sometimes had to be taken advantage of to make the benchmark work on a system. However, we think that the technology has matured since we did the experiments and expect it to become more robust so that a detailed report of these experience would probably be already outdated. We therefore just mention some findings and refer to [21] and [22] which contain more detailed material.

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 chapter. We anonymized the systems due to well-known license restrictions and we simply speak of systems A through F. These systems are designed as *large scale repositories* and therefore can be expected to perform well at handling large amounts of data. In the sequel, we will refer to these systems also 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.

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.

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 1.1. Database sizes

Concerning the scaling factor, all mass storage systems were able to process the queries 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 1.1: they range from 50 seconds to 781 seconds. They are 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 1.1; we remark that some systems which are not included in this comparison require far larger database sizes.

	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 1.2. Performance in ms of some queries

We now turn our attention to the running times and statistics as displayed in Table 1.2 and present some insights. Since we do not have the space to discuss all timings and experiments in detail, we only present a selection. In most physical XML mappings found in the literature, Query Q1 [18] 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 look-ups if a mapping like [20] is used.

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 1.3. Detailed timings of Q1 and Q2 for Systems A, B, C

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 representation [18] of the queries might suggest. Consequently, running times are quite high. Although System A was able to find an execution plan for Q3 which was as good as that of the other systems, it spent too much of its time on optimization. Table 1.3 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 (including 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 1.3, 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.

Query Q5 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 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 pre-calculation 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 since 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.

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 1.2 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. The aggregations of Q20 conclude the query set with a combination of three table scans and a set difference. All systems show similar performance.

In some of the performance figures certain systems (particularly Systems A to C) show pathological running times (*cf*. Table 1.2). 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 expressions 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.

1.6 Conclusion

In this chapter we outlined the design of XMark, a benchmark to assess the performance of query processors for XML documents. Based on a internet auction site as an application scenario, XMark provides a suite of queries that have been carefully crafted to highlight individual performance critical aspects inherent to the querying of XML. As work on the benchmark started at an early stage in the development of XML query processors, the philosophy behind it evolved to keep up with its targets. Since its release XMark has been widely adopted by both research communities and industry.

References

1. S. Abiteboul. Querying Semi-Structured Data. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 1–18, 1997.
2. 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.
3. T. Böhme and E. Rahm. XMach-1: A Benchmark for XML Data Management. In *Proceedings of BTW2001*, 2001.
4. A. Bonifati and S. Ceri. Comparative Analysis of Five XML Query Languages. *ACM SIGMOD Record*, 29(1):68–79, 2000.
5. R. Bourett. XML Database Products. available at `http://www.rpbourret.com/xml/XMLDatabaseProds.htm`, 2000.
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, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. Xml query use cases. Technical report, W3C, November 2002. available at `http://www.w3.org/TR/xmlquery-use-cases/`.
12. 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`.
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. Runapongsa, J. M. Patel, H. V. Jagadish, and S. Al-Khalifa. The michigan benchmark: A micro-benchmark for xml query processing system. Informal Proceedings of EEXTT02, electronic version available at `http://www.eecs.umich.edu/db/mbench/`, 2002.
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. XMark Queries, 2002. available at `http://www.xml-benchmark.org/Assets/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, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the International Conference on Very Large Data Bases*, pages 974–985, 2002.
22. A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI Amsterdam, 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. B. B. Yao, M. T. zsu, and J. Keenleyside. XBench - A Family of Benchmarks for XML DBMSs. Technical Report TR-CS-2002-39, University of Waterloo, 2002.