Processing XML in Database Systems
Schmidt, A.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 5

Performance of XML Query Processors

This chapter\(^1\) discusses issues that arise when the performance behaviour of XML processing systems is to be assessed and tuned. The first part discusses general issues that arise in benchmarks and points at the most prominent peculiarities of XML-related performance assessment in more detail. The second part then presents the XML Benchmark XMark, which was co-developed by the author and successfully deployed in a number of environments.

5.1 Why and How to Benchmark XML Databases

Benchmarks belong to the core repertoire of tools and methodologies deployed in database development. The assessment of the capabilities of a system, the analysis of actual and potential bottlenecks, and, naturally, the comparison of the advantages and disadvantages of different systems architectures have become indispensable tasks as databases management systems have started to grow in complexity, capacity, and versatility. In the course of the development of XML databases the need for a benchmark framework has become more and more evident: the many different ways to store XML data which have been suggested in the past each come with their genuine characteristics and consequently propagate through the various layers of a complex database system and, thus, need to be carefully considered. The different storage schemas render the query characteristics of the data quite different. However, no conclusive methodology for assessing these differences is available to date.

In the sequel, we outline desiderata for benchmarks for XML databases; we draw from our own experience of developing an XML repository with a query language and familiarity with standard benchmarks for relational databases.

5.1.1 Introduction

It may not be obvious, at first sight, why XML and database management systems should go together well: after all, they seem like two very different concepts driven by two very different communities that have different expectations and requirements

\(^1\)Parts of this chapter were published in [SWK\(^+\)01b, SWK\(^+\)01a, SWK\(^+\)02]
which do not necessarily overlap. Yet, there is an increasing demand for consistent and reliable ways of managing XML data [Bou00a, Bou00b] that suggests that the marriage of the two is beneficial. By that, we do not mean physical management of XML data on top of a relational database as it is often interpreted but a general coupling of the two worlds by aligning access protocols and environments. For storing XML, numerous techniques have been suggested, such as the ones proposed or described in this thesis, and the discussion of which database technology, be it relational or extended relational, object-oriented or native, is up to the challenge is still in full swing.

From the point of view of an application, the discussion whether XML is syntax or data model looks slightly different: in order to assume the role of a true communication framework, in which both industry and research want XML to be seen, an XML database has to deliver on the performance demands of its key applications, which are Web services, business-to-business (B2B), and electronic commerce scenarios to name just a few. Most of them require on-line, very often interactive processing of requests.

Throughout the history of relational database technology, benchmarks served primarily as a stake in the ground to assess and compare if and how new techniques and system components integrate into existing infrastructure and how they perform. On the other side, benchmarks cross-fertilised many innovations in query processing and storage management, which were achieved by trying to boost benchmark figures; for examples, see [GLJ01, GL01]. Thus, benchmarking does not only mean that we measure the state of the art but is also a constant incentive for further development. Here, we postulate desiderata for a general purpose benchmark for XML databases and take both into account, the state of the art as well as recent developments.

First of all, we have a look at the current state of the technology: (1) Currently, there is no commonly agreed on notion of the functionality that an XML database can be expected to comply with. Therefore, we will go with the frequently used definition of an XML database, which refers to a system that stores and manages XML documents. A number of similar definitions and a more in-depth discussion can be found at [Bou00b]. More crisp are the definitions of the various query languages such as Quilt's successor [CRF00a] XQuery [CFR+01], which have become available. (2) On the other hand, many applications which are currently under construction or are being deployed already specify a number of requirements which XML databases will have to meet. But before going into a detailed analysis, it is beneficial to take a look at related work and survey briefly what development relational systems and benchmarks have taken in order to identify reasonable criteria and to avoid obvious mistakes. Interesting question are what can be carried over from more traditional benchmarks like those defined by the Transaction Processing Council (TPC), SAP, PeopleSoft among others and what can be learnt from these efforts.

Early benchmarks have been strictly geared toward testing database functionality on a very general level. The Wisconsin Benchmark [Gra93] for example consists of a number of queries which test the performance of small numbers of join operations. While they help to determine the bottlenecks in join processing, the benchmark does not take a particular application scenario into account beyond the obvious motivation that joins rank among the most costly operations thus have to be catered for especially.

As database functionality evolved, implementations of different systems converged, and performance figures for simple queries became increasingly indistinguishable on different systems, the embedding of a database system into an application scenario became more and more important. Today, the TPC-H/R [PF00] are among the most important industry benchmarks; they are modelled after the general requirements which a data warehousing application poses to a database system.
5.1 Why and How to Benchmark XML Databases

Finally, the latest development are application specific benchmarks like the SAP [SAP01] and the PeopleSoft benchmark series [Peo01], where performance characteristics for running one proprietary application are used to assess a system. In contrast to the early general purpose benchmarks, these kind of application benchmarks no longer evaluate databases as isolated units but as an integrated back-end in an application scenario, and assume that customers purchase the database solely for this purpose.

This evolution may seem like a linear development with databases changing from general purpose repositories to becoming part of an individual, complex application. However, this development has also seen failures and set-backs, most notably TPC-W. The TPC-W benchmark is widely agreed to provide only little insight. While it models many components of an Internet-commerce application, the interaction between components and different scenarios become opaque and hard to analyse. Unlike the application benchmarks mentioned above, the TPC-W scenario is kept overly generic, and thus, instead of exactly matching one application, it does not match any. While tuning a database system for SAP's or PeopleSoft's benchmark is of direct use for customers who directly deploy the database system – often solely – in this scenario, the gains from high TPC-W numbers remains questionable.

In short, benchmarks have to match (1) the technology available at the time and (2) the application scenarios used in production. Given the current developments, defining a general purpose benchmark is an important first step to conclusive assessments of XML databases. We review the conceptual idiosyncrasies of storing and retrieving XML data and try to identify components and operations that a reasonable benchmark should cover. We also scrutinise what lessons learnt from query processing in relational databases directly carry over to the XML world.

The question 'How to assess query performance?' is obviously preceded by the more basic question 'What operations on an XML document are conceivable and reasonable?' With its XML Query Working Group, the W3C addressed this issue with experts from both the database and the document communities. They developed scenarios and use cases [CFMR01b] by taking into account both input from the research community and results of surveys of Web services and application scenarios that did not yet deploy XML but could benefit from it. The outcome of this process became available in two phases: first, an algebra that tried to formalise operations was released and later, a draft for a query language was formulated.

It appears essential, when staking out opportunities for a benchmark, to review these two products as they set the stage for any performance evaluation by defining the set of operations available.

5.1.2 Query Algebra

The algebra was specified in a high-level functional programming language [CFR+01] in the spirit of Haskell [Tho99]; it defined and illustrated by means of use cases a set of operations that appear meaningful to perform on XML documents. Due to the visible database background of the group, most operations suggested are very familiar from the world of relational database systems. Operations include filters, joins on values or along referential key constraints, grouping operations etc. The overall design of the algebra leads towards an implementation using comprehensions such as laid out in [GS99].

In contrast to their relational relatives, these operators obviously have to maintain order. In the notation of the algebra, this is implied by using for-loop like constructs which suggest iterating over an ordered list of data. These loops are only defining
the variable binding and a system which can encode order through, say, an additional attribute at the object itself may take advantage of using operations that are not order preserving, e.g., hash joins.

With respect to the definition of a benchmark, the algebra immediately provides two requirements. On the one hand, we certainly want to test most if not all operations suggested. On the other hand, the issue of treating order should be a central aspect in the tests—the more if the XML database in question is implemented on a relational database system.

5.1.3 Query Languages

While the algebra mainly served to outline the basic operations that would have to be supported by a language, it does not specify how those operations should be exposed in a query language. At the beginning of the standardisation process, several query languages were in use, the most popular of which are XQL [RLS98] and XPath [CD99]. Those languages might be considered obsolete in the light of the standardised query language XQuery – XQuery subsumes XPath and is more powerful than XQL. However, a sizable user base currently deploys alternative XML query languages including several proprietary ones and to the best of our knowledge no commercial implementation of XQuery is available at the moment. Since XQuery has been being standardised by the W3C, with all major manufacturers fully supporting the development, XQuery is clearly the query language of choice for formulating the benchmark queries. Additionally, the availability of the formal semantics of the language makes it even more attractive to be used in settings where verification of implementation is an issue.

Currently, not all XML stores provide full XQuery support – be it for reasons of integrating legacy applications and data, be it that the application scenario the XML database has been developed for simply does not require the additional expressiveness of XQuery. However, even without full XQuery they are interesting systems from which a lot can be learned; so, we still want to be able to test them. Thus, it seems highly advisable to define query groups, i.e., a classification and bundling of the individual queries. Often, simple queries can be formulated using only XPath primitives. A second dimension along which we will define query groups is, of course, the functionality. That way, the functionality of a system can be categorised and assessed conclusively and transparently.

The XQuery standardisation efforts have come a long way already, yet, XQuery is by no means complete. So far merely read-only query scenarios have been addressed; inserts, updates and XML specific derivatives thereof have, at the time of writing, not been dealt with. The organisation of the benchmark into query groups also helps in this situation as groups can be enhanced with versioning to reflect the current standard. Releasing a new version of a particular query group to incorporate new developments does not invalidate old results.

5.1.4 Motivating Examples

For XML data the physical breakdown of data is of significant impact and decidedly more important than in the relational world. The decision how to store a document not only depends on performance and redundancy considerations but also on knowledge external to the document in the form of constraints. In the simplest case, there is no external knowledge and one has to use a generic mapping like one of the binary mappings described or mentioned for instance in Chapter 3. If there is external knowledge,
5.1 Why and How to Benchmark XML Databases

be it in the form of constraints like a DTD or even something more expressive like a schema language [LC00], then one may decide to use a more advantageous mapping for example by inlining 1:1 relationships into larger relations or mapping generic string values to richer data types to save space and avoid coercions at query execution time. In an object-oriented scenario, where the primary access pattern is navigational iteration over DOM like syntax trees [W3C98a], it is often worthwhile to map the document structure to a set of class definitions. In this case, external knowledge can be exploited by regrouping objects into disk pages according to different access scenarios. Depending on the application scenario there likely are more optimisations to favour the query profile of the application. Although numerous approaches to XML mapping have been proposed in the literature, it seems that two design principles prevail: binary decomposition, i.e., grouping ancestor relationships in one or more relations, and inlined representation, i.e., storing 1:1 relationships between parent tags and their element and attribute children in one relation while 1:n relationships are mapped to two relations which share a common key. While the former excel through their conceptual simplicity, the latter try to retain a relational flair by mimicking the translation rules (i.e., fragmenting the document along 1:n parent-child relationships) between ER-models and relational schema.

Closely related to the issue of the physical representation is that of query optimisation. Over the past two decades, there have been significant advances in optimising SQL queries; in practise rule-based and cost-based optimisation techniques are used, both of which are tightly coupled to the underlying physical data model. It would be highly desirable to re-use large parts if not all of this knowledge. For cost-based techniques, sampling methods provide estimates to search and prune large spaces effectively. Sampling methods are the back-bone of most modern query optimisers, are designed for common query profiles such as (equi-)joins on key attributes and have been successfully deployed. Yet it is not clear how good a job they do in the case of hierarchically structured and untyped XML documents or whether they transfer to native XML query processors.

To illustrate these points, we implemented different mapping models on top of a relational database system and compared the query times for three queries taken from The XML Benchmark Project presented later in this chapter. Figure 5.1 shows the resulting query times: The first column in each group represents the time for a fixed, document-independent binary storage model such as [MAG+97a], i.e., all parent-child relationships are stored in a single relation resulting in many self-joins during query execution. The second column is the performance of the same data mapped to a variable, document-dependent relational module such as [SKWW00a], i.e., all parent-child relationships of a specific type are clustered in a separate relation. The third column in each group shows the performance of an inlined multi-attribute mapping in the spirit of [STH+99]; here relations are larger and can have many more attributes than there are per relation in the binary case. From the plot, it is evident that each of the models has its advantages and can outrun all other alternatives in certain situations. However, to understand the issues and bottlenecks encountered in this simple experiment we need to have a closer look at the processing of the queries:

Query 1 is a very basic point query specified by a path expression. On the inlined model this results in a scan of one relation returning a single tuple; on the other models a number of joins or self-joins are required to materialise the path expression. On the one hand, the inlined model trades in data volume for joins with respect to the binary models. On the other hand, in the variable model a larger number of tables has to be accessed.
Query 2 specifies a join depending on the document order. Here the multi-attribute model benefits from the fact that fewer joins need to be executed than in the binary case. As to the other two models, the implicit clustering of data as achieved by the variable model results in better statistics as compared to the fixed model. Therefore, the variable model is significantly more amenable for the optimiser than the fixed model.

Query 3 finally underlines the differences in numbers of columns per table. The multi-attribute model suffers from the data volume when a large number of joins is executed where only few columns are required for the result. Joins are much less costly to process on both the fixed and the variable decomposition schemes.

One goal of the XML Benchmark Project is to provide a framework which is more complete than the sketchy example presented here and help those who interpret the benchmark results to get an impression of what impact design decisions possibly have. The example provides a good impression of the extent and nature of the performance issues a benchmark has to address. Besides the relatively obvious question of the physical data break-down, a number of other aspects have severe impact on the query performance. Concerning the physical design, the usage of indexes and surrogate OIDs where additional information like order is encoded in the OID can speed up querying at the expense of rendering updates or inserts overly expensive. An area which has been largely neglected so far is the optimisation of XML specific queries. Besides issues like in Query 2 where the physical storage scheme prevents extracting meaningful statistics, relational optimisation techniques are also oblivious of XML specifics like constraints imposed by the hierarchical structure of the document. In term of optimisation, it may be necessary to extend the optimiser logic with additional rules and search heuristics. Similarly, the question of how to deal with intermediate results or XML views needs to be addressed. It is necessary or useful to materialise them? If yes, according to the physical, logical or XML model? Finally, a related question: What is the best strategy for converting a database or intermediate result to an XML document

Figure 5.1: Execution times of select benchmark queries on a relational database system using three different mappings of XML to relational structures
on what architecture (see [SSB+00] for the relational case)?

5.1.5 Challenges

Above, we explained some of the effects that caused significant differences with respect to performance characteristics though the physical storage format differed only slightly and was based on the same relational database architecture. Certainly, a general performance assessment cannot scrutinise differences on as fine a level of granularity as we hinted at. Rather we identify ten general challenges which aim at providing a comprehensive and conclusive performance analysis covering all performance critical aspects of processing of XML.

1. **Bulk loading.** The importance of bulk loading data has been repeatedly emphasised when assessing database performance in general. In XML databases, bulk loading currently assumes an even more important role as no insert/update operations are available and most systems support insert only on a document level. Due to the fact that different models imply different levels of granularity when shredding the document this seemingly simple operation may entail severe costs to setup and maintain indices or constraints.

2. **Reconstruction.** Also known as round-tripping, reconstructing the original document is the counterpart of bulk loading. Though being an operation simple to specify, round-tripping is not a common but still necessary operation in most scenarios. Reconstruction reveals the price of achieving lossless storage of the document, i.e., the trade-off between efficient indexing and preserving the full semantics of the document.

3. **Path traversals.** Specifying paths arguably is one of the most basic and natural operations on structured documents. Not only useful as a stand alone operation, path expressions are furthermore a ubiquitous building block of almost all complex operations. Efficient path traversals often bring about a trade-off with respect to redundancy, data volume or degree of fragmentation.

4. **Casting.** XML is essentially text and as such queries frequently demand casting to other elementary data types like integers, floats or even user-defined types. String operations are notoriously expensive.

5. **Missing elements.** The semi-structured nature of XML in general brings about a highly heterogeneous structure of records that under some mappings results in many NULL values. Apart from strategies to store NULL values in a compact way, we also need efficient methods to query for NULLs as they often represent spots of interest.

6. **Ordered access.** Order is an omni-present feature when querying XML and affects all aspects of data management. Obviously, sticking with order preserving implementations may become a severe bottleneck. Rather, sophisticated and flexible treatment of the document order should ensure that it well integrated into the optimisation process up to the degree that it is ignored when not needed, which could be specified by the user.

7. **References.** References are an important modelling primitive almost comparable to referential constraints in relational databases. In general, chasing references
requires efficient access methods supporting random access across the document rather than navigational access.

8. **Joins.** Join operations on values, *i.e.*, content, have been seen as a critical aspect early XML query languages lacked. Particularly data-centric applications require the combination of data based on values. The difficulties, bottlenecks, and challenges posed by joins are well-known from relational database systems.

9. **Construction of large results.** As opposed to round-tripping, construction of large results refers to assembling large new documents from the data stored. Many application areas demand the ability to handle large data volumes.

10. **Containment, full-text search.** Containment and full-text search are elementary operations when querying XML. The problem and its intrinsic difficulties are well-known from other application domains, hence, it hardly needs further motivation.

A benchmark for XML databases needs to address these ten points. Given an application scenario, most of the challenges can be directly expressed as a single query, some however offer a larger degree of freedom and it seems advisable to address them with several, differently parameterised queries.

### 5.1.6 Other Quality Parameters

The above challenges clearly focus on the performance of the XML database in isolation as motivated in the Introduction. From an application programmer’s perspective, other quality parameters of an XML database are visible and at first sight, it may seem natural to include those. They can be divided into two groups, infrastructure issues and total cost of ownership. We list them here for completeness and underline afterwards why we believe they should not be included in the list of challenges.

**Infrastructure**

Especially if front-end and back-end of the database are not tightly coupled, communication costs may dominate and obscure the performance characteristics of the query processor. Some of these issues have been addressed in XMach-1 [BR01], a mainly system-focused benchmark. Important issues include:

- **Access protocols** like HTTP, OLE DB, ODMG, ODBC, native APIs *etc.*, including their noise and transmission costs, often determine the degree of usability of a solution.

- **Result representations** like DOM, SAX, serialised XML, partly or fully materialised in standard or proprietary structures should be in line with application requirements.

- **Responsiveness versus completeness, i.e.*, availability of the first or the complete query result, including the influence of lazy evaluation and the availability of cursors can have a great impact on performance of production systems;

- **The expressiveness of the query language, e.g.*, the missing restructuring capability of XPath queries, often determines whether a query engine fits a given application scenario;
Lastly, *data throughput* in multi-user application scenarios;

One certainly could come up with a much longer and more detailed list, which would be beyond the scope of this thesis, but what it shows is that real-world application not only require high query throughput in the core engine but that overall performance and architecture considerations are necessary. Even for the purpose of tuning systems however, analysis of query processor core can still prove beneficial.

Total Cost of Ownership

With increasing complexity of software systems, the total cost of ownership becomes more and more important as it usually dominates the costs of the software itself. With respect to XML processing, we can identify a number of issues:

- **Installation effort:** Does the product work out of the box or does it require extensive multi-stage preparation and installation?
- **Generality support:** Is it possible to store documents with and without schema information? What price does one have to pay for lack of modelling?
- **Consistency support:** Is it possible to validate incoming documents against a schema or other constraints?
- **Preparation effort:** What mapping and schema definition tasks are necessary before the document can be imported.
- **Training:** Does the Software require extensive training for technical staff or does it integrate into existing infrastructure?
- **Interaction paradigm:** Does the product provide a stand-alone document management system with tools for direct interaction, or is it rather an enabling technology for front-end applications?
- **Updates:** Does the system provide fine-grained update functionality or is it restricted to replacing complete documents?

For practitioners, the ease of use of the installed product is often a most decisive factor. The ability of the system in question to adapt to the hardware infrastructure like disk type and topology, multiprocessor architectures, firewalls as well as accessibility through the software of choice, be it CGI, Web servers, Servlets or direct access.

In the Introduction we already outlined why not to benchmark infrastructure issues or total cost of ownership: A benchmark has to reflect primarily the state of the art. Given the current developments, we believe a benchmark that addresses the performance challenges we listed above provides the most valuable input for the advancement of XML processing technology. Benchmarks which include other aspects provide less accurate and insightful feedback than we envision.

### 5.1.7 Conclusion

The wide choice of architectures and environments makes it difficult to decide which XML Query Processor and which infrastructure fit best a given application. From a systems analysis point of view, the noise introduced by the infrastructure into which the query engine is embedded often obscures the performance of the core components
and makes fine-tuning and architectural improvements hard to realise. Therefore, the XML Benchmark Project, in which we gathered our experiences, aims at providing tools to analyse and improve query processors and make sense of their performance characteristics. We have motivated the need for an XML benchmark by pointing out which parts of systems need consideration and adaptation: while the front-end query language is in the process of standardisation, there are many performance critical issues in the design of the physical storage schema, the physical algebra, query optimiser and execution. The desiderata listed in this chapter are intended to serve as a basis for a comprehensive performance analysis of XML databases.

5.2 The XML Benchmark XMark

This section presents a benchmark that is intended to fulfil the requirements outlined in the previous section. The project was started very early in the history of XML just when it became apparent that users would be interested in storing large amounts of XML documents in databases. Since then it has evolved in line with the development of XML. The benchmark has been deployed in practise and is used in a number of research projects and to optimise and evaluate commercial products.

5.2.1 Introduction

While standardisation efforts for XML query languages have been progressing, researchers and users increasingly focus their attention on the database technology that has to deliver on the new challenges that the sheer amount of XML documents produced by applications poses to data management: validation, performance evaluation and optimisation of XML query processors are the upcoming issues. Following a long tradition in database research, we provide a framework to assess an XML database's abilities to cope with a broad spectrum of different query types which are typically encountered in real-world application scenarios. The benchmark is intended to help both implementors and users of database technology to compare XML databases independent of their own, specific application scenario. To this end, the benchmark offers a set queries each of which is intended to challenge a particular aspect of the query processor or storage engine.

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 query processing. The queries' challenges range from stressing the textual character of the document to data analysis queries, but include also typical ad-hoc queries. We complement our research with results obtained from running the benchmark on several XML database platforms. They are intended to give a first baseline and illustrate the state of the art.

The data exchange format XML has been penetrating virtually all areas of Internet application programming. Thus, content providers 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 organising data lacking a fixed schema, which seemed incompatible with existing technology at first sight. However, as XML gains 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 break-down and query performance have started to determine the success or failure of the solutions under development.

Currently database vendors (see [Bou00a] for an 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 is a primarily textual markup language, which means that unlike in the case of (O)RDBMS, data elements are ordered by nature; string is the core data type, from which richer data types, i.e., integers, floats and even user-defined abstract data types are derived; externally provided schema information, which may or may not be present, helps 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, i.e., IDs and IDREFs, 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 map this wealth of new properties to existing data models undoubtedly provides 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, e.g., see [RPNK00]. Due to their complexity, interaction, and interdependencies with various system components, most of the designs, with their obvious advantages and disadvantages, are hard to assess without putting them to the only conclusive test: a comprehensive quantitative assessment, or in short the right benchmark.

Dealing with design decisions as outlined above is certainly no new issue 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 Benchmark [ABM+90], the object-oriented 001-Benchmark [CS92], OO7-Benchmark [CDN93] or the object-relational BUCKY benchmark [CDN+97] to standards in industry like the family of TPC benchmarks [Gra93, Tra02] just to mention some of the better known examples. However, none of the available benchmarks offers the coverage needed: all of them are geared towards a certain data model but the flexibility and possibilities of semi-structured query languages exceed existing systems’ limitations by far.

The XMark Benchmark Version 1.0 described in this section 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 document database and a comprehensive set of queries. 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 feature-wise examination of the query processors proves beneficial as a query 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 rela-
tional, object-relational, main-memory and object-oriented database technology as well from textual information retrieval data structures and persistent object stores. Therefore, different products can be expected to display diverging behaviour in performance and stress tests according to their system architecture. Second, the benchmark document and the queries can aid in the verification of query processors, which has been a challenging problem ever since high level query languages were introduced [Slu98]; since we are now in the early days of XML bulk storage this is of special interest. In the world of markup languages, the problem of equivalence of XML documents, which XML query processors are expected to output, gets worse. The degrees of freedom the different possible physical representations of the document (see [Boy01] for an attempt to tackle it) introduce, are combined with the degrees of freedom in query execution regarding order of set-value attributes, different character encodings, namespaces etc. In our experience, there it is still an open research question when one should regard the output of XML query processors as equivalent. Third, executing the benchmark query set exhibits details of the workflow 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 fit 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 even transformations (e.g., XSLT) of the output. Our test scenario is as depicted in Figure 5.2. As for the choice of language, we use XQuery [CFR+01], an amalgamation of many research languages for semi-structured data or XML (for an overview see [BC00]) and a proposed future standard. It is in the process of standardisation 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.

So, the target audience of the benchmark can be divided into three groups. First, the framework presented here can help other 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 still 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 of existing or design of new algorithms.

The rest of the section 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 preliminary results obtained by running the queries in our test environment.

5.2.2 Motivation

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

Not surprisingly, the evolution of XML differs significantly from the evolution of relational databases in that there was an agreed 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, we think that 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 represent data structures, map very nicely to relational databases or object-relational databases, it is less clear how the same systems can handle efficiently documents that are 'text-oriented', i.e., natural language with markup interspersed (e.g., compare the performance of Google's [BP98] full-text search with that of your favourite RDBMS + XML mapping combination). Therefore we want to give hints how different DBMS architectures respond to the XML challenge which can be summarised as follows:

- 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 5.2.5).

- Strings are the basic data type; they can vary greatly in length posing additional storage problems. Type mismatches are likely to arise when the typing rules of query or programming languages clash with the generic string tokens of XML.

- 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 on relational systems.
To compound matters, the *loose schema* of many XML documents can make query formulation tedious from a user’s point of view. Technically, NULL values can blow up the size of the database. From a user’s viewpoint, specifying long and complicated path expressions is error-prone.

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

**5.2.3 Database Description**

An XML Benchmark requires a carefully modelled database to make the behaviour 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.

**Hierarchical Element Structure**

The nesting of elements renders the overall tree structure of XML documents. In this subsection we describe the structure of the document which is modelled 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 5.3. The semantics of the entities just mentioned is as follows:

![Element relationships between most of the queried elements](image)
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 over time) along with references to the bidders and the seller, the current bid, the default increase, the type of auction (Dutch, Featured, Regular), the time interval within which bids are accepted, the status of the transaction and a reference to the item being sold.

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 amount 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 characterised 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 watch.

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. This is in stark contrast to the offspring of *annotation* and *description* elements which make up the dual nature, document-centric side, of XML. Here the length of strings and the internal structure of sub-elements varies greatly. The markup now comprises itemised 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. The appendix gives an impression of the document by showing some snippets. An Entity-Relationship diagram of the database can be found in [BDL*01].
References

An overview over the references that connect sub-trees is given in Figure 5.4. Care has been taken that the references feature diverse distributions, derived from uniformly, normally and also exponentially distributed random variables. Note that all references are de facto typed, i.e., all instances of an XML element point to the same type of XML element; for example, references of interest objects always refer to categories.

Generated Text

To generate text that bears the characteristics of natural language text, we analysed Shakespeare's plays and determined statistical characteristics of the text. The generator mimics this distribution using the 17000 most frequently occurring words of Shakespeare's plays. We did not incorporate additional characteristics like punctuation as it is only of little relevance for the performance assessment. We believe tokenisation and other text compression methods commonly used can be sufficiently well assessed with the text we provide.

For names, email addresses etc. we used various sources like electronically available phone directories etc. Care has been taken to preserve important characteristics of the original material, and at the same time to scramble data like names by combining different first and last names. Similar techniques were used to avoid clashes with existing email addresses and so forth.

XML Constructs

The XML Standard [BPSMM00] defines constructs that are useful for producing flexible markup but that we think do not justify the definition of separate query challenges. Therefore, we only made use of a restricted set of XML features in the data generator which we consider essential to XML processing. For example, we do not generate documents with Entities or Notations. Neither do we distinguish between Parsed Character Data and Character Data as we assume that both are just string types from the viewpoint of the storage engine. Furthermore, we don't include Namespaces in the queries. We also restrict ourselves to the seven bit ASCII character set.

We provide a DTD and XML Schema information to allow for more efficient mappings. However, we stress that this is additional information that may be exploited.

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

- *platform independent* so that any user interested in running the benchmark is able to download the binary and generate exactly the same document no matter what hardware or operating system is used;

- *accurately scalable* ranging from a minimal document at scaling factor zero 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 custom random number generator rather than relying on the system’s built-in random number generators. Together with basic algorithms which can be found in statistics textbooks this unit implements uniform, exponential, and normal distributions of fairly high quality. We assigned 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 organised 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. Figure 5.5). In our tests, we found the resulting document size to deviate from the anticipated one usually by less than 1 percent. Finally, meeting the efficiency goals is an interesting challenge: many references are created at different places and the straightforward solution – namely, keeping some sort of log that records which IDs have already been referenced – is 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 can be found at [SKF+00].

5.2.4 Bulkloading the Document

In the context of XML, the rôle of bulkloading stands apart from its importance in other benchmarks. Since (prospective) standards like XQuery are not explicitly designed as database but data integration languages, we can, strictly speaking, assume neither 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. Whereas previous XML benchmarking efforts [BR01] are designed as application-centric multi-user workloads and include updates and bulk operations, we dispense with doing so because to date there is neither a standard nor an agreement on the exact semantics of updates. We therefore resort to the following interpretation: We take the XQuery Syntax for

\[
\text{FOR } a \text{ in document("auction.xml")/...}
\]
literally and formulate all queries with respect to a single large document without committing ourselves to a specific database scenario.

Although we did not experience problems when bulkloading the document in our test environments, we 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 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 respectively IDREFs. 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 an 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 are normative.

### 5.2.5 Benchmark Queries

This section lists the benchmark queries. We chose to express the queries in XQuery [CFR*01], the successor to the research language Quilt [CRF00b], which is about to be standardised.

#### Exact Match

This simple query is mainly used to establish a simple performance primitive unit to help establish a ‘metric’ to interpret subsequent queries. It tests the database’s ability to handle simple string lookups with a fully specified path.

**Q1. Return the name of the person item with ID ‘person0’**

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

#### Ordered Access

These queries should help users to gain insight into 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.**

```
FOR $b IN document("auction.xml")/site/open_auctions/open_auction
RETURN <increase> $b/bidder[1]/increase/text() </increase>
```

This query evaluates the cost of array look-ups. Note that this query 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.**

```
FOR $b IN document("auction.xml")/site/open_auctions/open_auction
WHERE $b/bidder[0]/increase/text() * 2 <=
```
This is a more complex application of index lookups. In the case of a relational DBMS, the query can take advantage of efficient implementation of set-valued aggregates on the index attribute to accelerate the execution. Queries Q2 and Q3 are akin to aggregations in the TPCD [Gra93] benchmark.

**Q4. List the reserves of those open auctions where a certain person issued a bid before another person.**

```
FOR $b IN document("auction.xml")/site/open_auctions/open_auction
  WHERE $b/bidder/personref[id="person18829"] BEFORE $b/bidder/personref[id="person10487"]
RETURN <history>$b/reserve/text()</history>
```

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

**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 in isolation. 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?**

```
COUNT
  (FOR $i IN document("auction.xml")/site/closed_auctions/closed_auction
   WHERE $i/price/text() >= 40
   RETURN $i/price)
```

**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 optimise path expressions and prune traversals of irrelevant parts of the tree.

**Q6. How many items are listed on all continents?**

```
FOR $b IN document("auction.xml")/site/regions
RETURN COUNT ($b/item)
```

A good evaluation engine or path encoding scheme should help realise 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?**

```
FOR $p IN document("auction.xml")/site
  LET $c1 := count($p//description),
     $c2 := count($p//mail),
     $c3 := count($p//email),
     $sum := $c1 + $c2 + $c3
RETURN $sum
```

Also note that the COUNT aggregation does not require a complete traversal of the tree. Just the cardinality of the respective relation is queried. Note that the tag `<email>` does not exist in the database document; hence $c3 evaluates to zero.
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 optimiser should take advantage of the cardinalities of the sets to be joined.

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

```xml
FOR $p$ IN document("auction.xml")/site/people/person
LET $a :=
    FOR $t$ IN document("auction.xml")/site/closed_auctions/closed_auction
    WHERE $t$/buyer/@person = $p$/@id
    RETURN $t$
RETURN <item person=$p$/name/text() COUNT ($a) </item>
```

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

```xml
FOR $p$ IN document("auction.xml")/site/people/person
LET $a :=
    FOR $t$ IN document("auction.xml")/site/closed_auctions/closed_auction
    LET $sn :=
        FOR $t2$ IN document("auction.xml")/site/regions/europe/item
        WHERE $t2$/itemref/@item = $t2$/@id
        RETURN $t2$
    WHERE $p$/@id = $t$/buyer/@person
RETURN <item $sn$/name/text() </item>
RETURN <person name=$p$/name/text() $a </person>
```

Construction of Complex Results

Constructing new elements may put the storage engine under stress especially in the context of creating materialised document views. 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 markup into French.

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

```xml
FOR $i$ IN DISTINCT
document("auction.xml")/site/people/person/profile/interest/@category
LET $sp :=
    FOR $t$ IN document("auction.xml")/site/people/person
    WHERE $t$/profile/interest/@category = $i$
RETURN
    <personne>
        <statistiques>
            <sexe> $t$/gender/text() </sexe>,
            <age> $t$/age/text() </age>,
            <education> $t$/education/text() </education>,
            <revenu> $t$/income/text() </revenu>
        </statistiques>,
        <coordonnees>
            <nom> $t$/name/text() </nom>,
            <rue> $t$/street/text() </rue>,
            <ville> $t$/city/text() </ville>,
            <pays> $t$/country/text() </pays>,
            <reseau>
                <courrier> $t$/email/text() </courrier>,
```
[FMS01] point out that the cost of generating textual answers and creating materialized views may differ greatly. Note that we do not require the system to construct a materialized view. However, comparing the relative costs of creating a view versus outputting text is certainly interesting.

**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 optimised with logical OIDs for example. The two queries Q11 and Q12 cascade in the size of the result set and provide various optimisation opportunities. We note that alternative query formulations with the '->' operator may be used.

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

```xml
FOR $p IN document("auction.xml")/site/people/person
LET $1 :=
  FOR $i IN
document("auction.xml")/site/open_auctions/open_auction/initial
WHERE $p/profile/@income > (5890 * $i/text())
RETURN $1
RETURN

<items>
  <name> $p/name/text() </name>
  <number> COUNT ($1) </number>
</items>
```

**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.

```xml
FOR $p IN document("auction.xml")/site/people/person
LET $1 :=
  FOR $i IN
document("auction.xml")/site/open_auctions/open_auction/initial
WHERE $p/profile/@income > (5000 * $i/text())
RETURN $1
WHERE $p/profile/@income > 50000
RETURN

<items>
  <name> $p/name/text() </name>
  <number> COUNT ($1) </number>
</items>
```
Reconstruction

A key design for XML→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.

Q13. List the names of items registered in Australia along with their descriptions.

FOR $i$ IN document("auction.xml")/site/regions/australia/item
RETURN <item> <name> $i/name/text() </name> $i/description </item>

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 markup is essential as the concepts are considered orthogonal; so query Q14 is restricted to a subset of the document by combining content and structure.

Q14. Return the names of all items whose description contains the word 'gold'.

FOR $i$ IN document("auction.xml")/site//item
WHERE CONTAINS ($i/description,"gold")
RETURN $i/name/text()

Path Traversals

In contrast to Section 5.4 we now try to quantify the costs of long path traversals that don’t include wildcards. We first descend deep into the tree (Query 15) and then return again (Query 16). Both queries only check for the existence of paths rather than selecting paths with predicates.

Q15. Print the keywords in emphasis in annotations of closed auctions.

FOR $a$ IN
document("auction.xml")/site/closed_auctions/closed_auction/
annotation/description/parlist/listitem/parlist/listitem/
text/emph/keyword/text()
RETURN <text> $a/text

Q16. Confer Q15. Return the IDs of the sellers of those auctions that have one or more keywords in emphasis.

FOR $a$ IN document("auction.xml")/site/closed_auctions/closed_auction
WHERE NOT EMPTY ($a/annotation/description/parlist/listitem/parlist/
listitem/text/emph/keyword/text())
RETURN <person id=$a/seller/@person />

Missing Elements

This query tests how well the query processors can 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?

FOR $p$ IN document("auction.xml")/site/people/person
WHERE EMPTY($p/homepage/text())
RETURN <person> $p/name/text() </person>

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

This query puts the application of user defined functions (UDF) to the test. 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.

**Q18. Convert the currency of the reserve of all open auctions to another currency.**

```xml
FUNCTION CONVERT (Sv)

RETURN 2.20371 * Sv -- convert Dfl to Euro

FOR $i IN document("auction.xml")/site/open_auctions/open_auction/
RETURN CONVERT($i/reserve/text())
```

Sorting

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

**Q19. Give an alphabetically ordered list of all items along with their location.**

```xml
FOR $b IN document("auction.xml")/site/regions/item
LET $k := $b/name/text()
RETURN
  <item>
    <name> $k </name>
    <location> $b/location/text() </location>
  </item>
SORTBY (. )
```

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.**

```xml
<result>
  <preferred>
    COUNT (document("auction.xml")/site/people/\n      person/profile[@income >= 100000])
  </preferred>,
  <standard>
    COUNT (document("auction.xml")/site/people/\n      person/profile[@income < 100000 and @income >= 30000])
  </standard>,
  <challenge>
    COUNT (document("auction.xml")/site/people/\n      person/profile[@income < 30000])
  </challenge>,
  <na>
    COUNT (FOR $p in document("auction.xml")/site/people/person
      WHERE EMPTY($p/@income)
      RETURN $p)
  </na>
</result>
```
5.2.6 Experiments and Experiences

The benchmark has been a group-design activity of academic and industry researchers and is known to be used to evaluate progress in both commercial and research settings. The evaluation presented here is meant to be an illustrative case study on a broad range of systems; an in-depth analysis would be beyond the scope of this work. We gathered our experiences with a broad range of systems, which we anonymised due to well-known licence and other restrictions; instead, we simply speak of systems A through J.

Broadly speaking, our test platforms fall into two categories:

- Systems that are designed as *large scale repositories* and therefore can be expected to 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 commercial relational technology and come with a cost-based query optimiser and allow the same kind of hand-optimisation 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 optimised database schema. System D shows the performance of Monet XML; Systems E and F are built on top of the main-memory based database system MySQL, which comes with a heuristic query optimiser; however, on Systems E and F we re-wrote queries by hand if the default execution plan did not terminate.

- Query processors that are intended to serve as *embedded query processor* 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 optimisation, meta-data access or I/O wait; others only allowed for 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 [SKF00]. 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 here possibly with minor syntactic changes. All queries were run on machines equipped with 550 MHz Pentium III processors and SCSI hard disks; operating systems were Windows 2000 Advanced Server and Linux 2.4 respectively depending on what the packages require. Although the systems were all equipped with at least two processors, only one processor was used during bulk load and query execution.

Concerning the scaling factor, we were not able to run all queries on all systems at scaling factor 1.0 (116.5 MB of text) as we had intended to do. The mass storage systems were able to process the queries but the embedded systems failed to do so because they ran out of resources.

For the mass storage systems, with which we commence our evaluation, the benchmark document was generated at scaling factor 1.0 (without indentation). Note that the pure parsing time (without semantic actions, *i.e.*, callbacks uninitialised) was 4.9 seconds (user time on Linux including system time, disk I/O and CPU stalls; parser
was expat [ea01]). The bulkload times are summarised in Figure 5.8. They range from 49 seconds to 781 seconds. Note that System C needs a DTD to derive a database schema; the time to do this is not included in the figure, but is negligible. The resulting database sizes are listed in Figure 5.2.6; the authors remark that some systems which are not included in this comparison feature far larger database sizes.

<table>
<thead>
<tr>
<th>System</th>
<th>bulkload time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>414</td>
</tr>
<tr>
<td>B</td>
<td>781</td>
</tr>
<tr>
<td>C</td>
<td>548</td>
</tr>
<tr>
<td>D</td>
<td>50</td>
</tr>
<tr>
<td>E</td>
<td>96</td>
</tr>
<tr>
<td>F</td>
<td>215</td>
</tr>
</tbody>
</table>

Figure 5.6: Bulkload timings

<table>
<thead>
<tr>
<th>System</th>
<th>Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>System A</td>
<td>241</td>
</tr>
<tr>
<td>System B</td>
<td>280</td>
</tr>
<tr>
<td>System C</td>
<td>238</td>
</tr>
<tr>
<td>System D</td>
<td>142</td>
</tr>
<tr>
<td>System E</td>
<td>302</td>
</tr>
<tr>
<td>System F</td>
<td>345</td>
</tr>
</tbody>
</table>

Figure 5.7: Database sizes

We now turn our attention to the running times as displayed in the performance figures and present some basic insights. In most data breakdowns, Query Q1 consists of a table scan 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 [SKWW00a] is used. In the sequel, please note that some figures use logarithmic scales.
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 since 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. In some sense, the query plan of Q3 looks like two Q2-plans (with different results, of course) combined in a snowflake-like fashion; 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 optimisation. Q4 features the ‘BEFORE’ predicate which is implemented with a theta-join over the aforementioned attributes which store the global order. Some mappings like [ZND+01] 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 better running times.

Figure 5.9 displays some interesting characteristics that can be traced back to the physical mappings the systems use. System A basically stores all XML data in 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 much less meta-information to compile a query than System B, thus spending only half as much time on query compilation 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 eventually needs a DTD to derive a storage schema; this additional information helps to get favourable performance. Still in Figure 5.9, we also find the detailed execution times for Q2. They show that
5.2 The XML Benchmark XMark

Figures 5.11 and 5.12 show the performance of queries Q3 and Q5, and Q7 and Q8, respectively. The charts illustrate the execution times for each query, with the x-axis representing the system identifiers A through F, and the y-axis representing execution time in milliseconds.

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 now come to Query Q5 which tries to quantify the cost of casting 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 if that was necessary such as for Queries 3, 5, 11, 12, 18, 20. We did not apply any domain-specific knowledge; neither did the systems use schema information or pre-calculation or caching of casting results.

Regular path expressions are the challenge presented by queries Q6 and Q7 (we only present results for Q7 due to lack of space). Some systems keep structural summaries of the database and can exploit them to optimise traversal-intensive queries. Therefore, Q6 and Q7 are surprisingly fast since unnecessary traversals can be avoided; however, on systems without access to structural summaries, which effectively play the role of an index or schema, these queries can be much more expensive to execute. The problem that Q7 actually looks for non-existing paths seems to be solved by the structural summaries of the mass storage systems. For some systems, the cost of accessing the structural summary was very high and dominated query performance.
Queries Q8 and Q9 are usually implemented as joins. In the systems that we could analyse in detail, chasing the references basically amounted to executing equi-joins on strings. We were surprised how cheap Q8 and Q9 were 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 an 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 a point that really can put XML processing systems to the test as Q10 shows. 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. Good markup strategies like the one presented in [SSB+00] should help to keep the processing time low.

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 optimiser to pick a good execution plan and allows insights into how the data volume influence query and output performance. For Systems B and C, the optimiser chose an unfortunate execution plan. Hand-optimised versions of the query on Systems D through F show that much better plans exist.
5.2 The XML Benchmark XMark

Figure 5.15: Performance of queries Q13 and Q14

Figure 5.16: Performance of queries Q15 and Q16

Query 13 reconstructs a good portion of the original input document. Since the query involves many joins, the systems based on the plain relational algebra are confronted with very large search spaces and fail to produce a query plan with the original input. The performance of Query 14 largely depends on the presence of a full-text index so that some systems feature a very competitive performance whereas others are comparatively slow. For reasons of fairness, we did not include the timings for Systems A through C for Q13 and Q14 since we had to use the view mechanisms of the database engine to execute them: the large number of joins made it impossible to run them in their original form.

Up to now, we have not tried to estimate the influence of long path expression versus short ones. This is exactly where Q15 and Q16 can help. Performance figures are displayed in Figure 5.16. Naturally, Q15 as a ‘subset’ of Q16 executes, as could be be expected, in shorter time than Q16. This is due to the many joins that the more complicated path expression in Q16 brings about – both in execution and optimisation. Also, the multi-pass SQL output of System E needed some massaging to keep intermediate results small.

Q17 again stresses the loose schema of most XML documents by querying for non-existing data. The query execution plan computes the intersection of two sets. Queries Q18 and Q19 are primarily of interest to establish the relative costs of function application and sorting operations when comparing two system architectures. Q17 to Q19 perform favourably as the binary fragmentation of the data model incurs only little
data volume so that all computations easily fit into main memory. The aggregations of Q20 conclude the query set with a combination of three table scans and a set difference. Again, the data model allows for a plan with very little data volume.

A general note on the queries: Not only did we often have 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 that was generated by the ‘native’ XML engines; this was especially the case with Systems E and F. Systems A to C on the other hand hardly needed manual intervention. Secondly, we’d 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.

**Embedded Processors**

For comparison, Figure 5.18 presents the performance behaviour of an embedded query processor on document sizes 100 kB and 1MB, which were the largest sizes we could sensibly compare. On the smaller document, no query took longer than 5 seconds but none was faster than 2.5 seconds; this means that the implementations techniques used for the embedded processor incur a significant performance overhead compared to the mass storage systems. With respect to join queries one should note that the result sizes were very small in comparison to those of the mass storage systems. A further advantage of the embedded systems is that they usually allow more control over query execution.

A note on query formulation: we used XQuery [CFR+01] as novices to formulate the benchmark queries. After a short phase of getting used to the language, we had no problems with the language itself. However, we would like to mention one point that would facilitate query formulation enormously. If a query processor was able to validate path expressions on-line, *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.

We are also looking forward to a standardised query language being agreed upon, which could tremendously facilitate the exchange of ideas between interested parties.

We also think that the ‘classic’ gap between data-centric and document-centric doc-
In this section, we presented the specifications developed in the XML Benchmarking Project, a set of queries and some lessons learnt while executing the workload specification on a number of platforms. The benchmark was designed top-down taking the standardisation 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, ordered data and coercions between data types. It demonstrated that there is no single best evaluation strategy that crosses the dividing line between queries that stress the textual nature of the document, like order constraints or reconstruction queries, and those queries that stress the data-centric perspective of associative retrievals. Optimisation and meta-data access are other issues that are handled more or less well by the systems.

Schema information is often useful in schema and query optimisation, especially when it gives hints about document-centric characteristics such as unscoped references that do not go together well with databases in general. However it only alleviates the problem that there are often competing design rules a schema can adhere to depending on the application area.

We expect our work to continue and evolve in the future. Formally, important parts of a complete application scenario are still missing: the lack of update specifications being the most prominent one. We hope that continuing standardisation efforts will make it possible to incorporate updates in the future. As an aside, we remark that so
far the benchmark has been received well in the community and that it is in use in a number of development and research groups both for verification of query processors and as a framework for testing, debugging and improvement.

5.3 Conclusion

This chapter motivated the need for an XML benchmarking methodology and presented the XML benchmark XMark in detail. Since benchmarks belong to the standard set of tools for database users and also represent the common methodologies used in data processing, they establish points of references that are useful for users who wish to compare systems.

XML stores are built on a wide variety architectures, each with its advantages and disadvantages or actual and potential bottlenecks. This make the comparison of different systems difficult; therefore, it is sensible to extend the database benchmark methodology to the XML world since database management systems have started to grow in complexity, capacity, and versatility. Hence the first half of this chapter outlined desiderata for benchmarking XML databases built upon our experience with XML processing. The second half presented the XMark benchmark along with performance figures on several systems. Unlike as in the case of relational benchmarks, XML systems still exhibit great differences in performance. This demonstrates the great maturity of relational database products in comparison to newer XML databases.

5.4 Bibliographical Remarks

By now there are other benchmarks available that can be used to evaluate certain aspects of XML repositories or database systems. For example, XMACH-1 [BR01] is a benchmark under development 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. X007 [BDL*01] is the XML counterpart of the popular OO7 benchmark [CDN93]; it comprises an XML version of the original OO7 database against which reformulations of the original queries are run. It also features extensions to the benchmark which are tailored towards stressing the XML-specific features the query processors provide. Our work differs from them in that it aims at large-scale analytical XML processing (unlike [BDL*01]) and at the same time offers query challenges that are designed along the lines of XML query algebras (unlike [BR01]) thus helping to analyse and improve the underlying query processor rather than merely measuring systems performance.