Benchmarking Performance-Critical Components in a Native XML Database System

Karsten Schmidt, Sebastian Bächle, and Theo Härder

University of Kaiserslautern, Germany
{kschmidt,baechle,haerder}@cs.uni-kl.de

Abstract. The rapidly increasing number of XML-related applications indicates a growing need for efficient, dynamic, and native XML support in database management systems (XDBMS). So far, both industry and academia primarily focus on benchmarking of high-level performance figures for a variety of applications, queries, or documents – frequently executed in artificial workload scenarios – and, therefore, may analyze and compare only specific or incidental behavior of the underlying systems. To cover the full XDBMS support, it is mandatory to benchmark performance-critical components bottom-up, thereby removing bottlenecks and optimizing component behavior. In this way, wrong conclusions are avoided when new techniques such as tailored XML operators, index types, or storage mappings with unfamiliar performance characteristics are used. As an experience report, we present what we have learned from benchmarking a native XDBMS and recommend certain setups to do it in a systematic and meaningful way.

1 Motivation

The increasing presence of XML data and XML-enabled (database) applications is raising the demand for established XML benchmarks. During the last years, a handful of ad-hoc benchmarks emerged and some of them served as basis for on-going XML research [5, 29, 39], thus constituting some kind of XML "standard" benchmarks. All these benchmarks address the XDBMS behavior and performance visible at the application interface (API) and fail to evaluate and compare properties of the XDBMS components involved in XQuery processing. However, the development of native XDBMSs should be test-driven for all system layers separately, as it was successfully done in the relational world, too, before such high-level benchmarks are used to confirm suitability and efficiency of an XDBMS for a given application domain.

In the same way, only high-level features such as document store/retrieve and complete XQuery expressions were drawn on the comparison and adaptation of XML benchmark capabilities [21, 26, 31, 32]. They can be often characterized as "black-box" approaches and are apparently inappropriate to analyze the internal system behavior in a detailed way. This applies to other approaches which focused on specific problems such as handling "shredding" or NULL values efficiently, too.

2 Related Work

Selecting an appropriate benchmark for the targeted application scenario can be challenging and has become a favorite research topic especially for XML in the recent years. In particular, the definitions of XPath [36] as an XML query language for path expressions and of the Turing-complete XQuery [37] language, which is actually based on XPath, caused the emergence of several XML benchmarks.

One of the first XML benchmarks is *XMach-1* [5] developed to analyze web applications using XML documents of varying structure and content. Besides simple search queries, coarse operations to delete entire documents or to insert new documents are defined. Thus, XMach-1 does not address, among other issues, efficiency checking of concurrency control when evaluating fine-grained modifications in XML documents. Another very popular project is *XMark* [33], providing artificial auction data within a single, but scalable document and 20 pre-defined queries. Actually such a benchmark is useful to evaluate search functions at the query layer, whereas, again, multi-user access is not addressed at all. A more recent alternative proposed by IBM can be used to evaluate transactional processing over XML – *TPoX* [29], which utilizes the *XQuery Update Facility* [38] to include XML modification operations. By referring to a real-world XML schema (FixML), the benchmark defines three types of documents having rather small sizes ($\leq$ 26 KB) and 11 initial queries for a financial application scenario. The dataset size can be scaled from several GB to one PB and the performance behavior of multi-user read/write transactions is a main objective of that project. Nevertheless, all benchmarking is done at the XQuery layer and no further system internals are to be inspected in detail.

However, other benchmarks explicitly investigate the XQuery engine in a stand-alone mode [1, 23]. Such "black box" benchmarks seem to be reasonable for scenarios operating on XML-enabled databases having relational cores, e.g., [4, 10, 11, 35], because their internals are optimized during the last 40 years.

Furthermore, it is interesting to take a look at the list of performance-critical aspects suggested by [31] and [32]: *bulk loading*, *document reconstruction*, *path traversal*, data type *casting*, *missing elements*, *ordered access*, *references*, value-based *joins*, construction of *large result sets*, *indexing*, and *full-text search* for the containment function of XQuery. Almost all aspects are solely focusing on the application level or query level of XML processing, or they address relational systems, which have to cope with *missing elements* (NULL values). Indeed, the development towards native XML databases, e.g., [15, 18, 28, 30, 34] was necessary to overcome all shortcomings of "Shredding" [35] and lead to new benchmark requirements and opportunities [27].

Other comparative work in [1] and [26] either compared existing XML benchmarks or discussed the cornerstones of good XML benchmarks (e.g, [21, 32]) defining queries and datasets. The only work that tried to extend existing approaches is presented in [31]. Although the authors forgot to consider TPoX, which has a focal point on updates, they extended other existing benchmarks with update queries to overcome that drawback.

3 Performance-Critical Components

To explore the performance behavior of DBMS components, the analyzing tools must be able to address their specific operations in a controlled way, provoke their runtime effects under a variety of load situations, and make these effects visible to enable result interpretation and to conduct further research regarding the cause of bottleneck behavior or performance bugs. The characteristics of the XML documents used are described in detail in the next section. Because the component analysis always includes similar sets of node and path operations, we do not want to repeat them redundantly and only sketch the important aspects or operations leading to optimized behavior.

XTC (XML Transaction Coordinator), our prototype XDBMS, is used in all measurements [15]. Its development over the last four years accumulated substantial experience concerning DBMS performance. Based on this experience, inspection of the internal flow of DBMS processing indicated the critical components benchmarked, which are highlighted in the illustration of the layered XTC architecture depicted in Fig. 1. Many of the sketched components have close relationships in functionality or similar implementations in the relational world. Therefore, stable and proven solutions with known performance characteristics exist for them. However, a closer study of the components highlighted reveals that their functionality and implementation exhibit the largest differences compared to those in a relational DBMS. Therefore, it is meaningful to concentrate on these components and their underlying concepts to analyze and optimize their (largely unfamiliar) performance behavior.

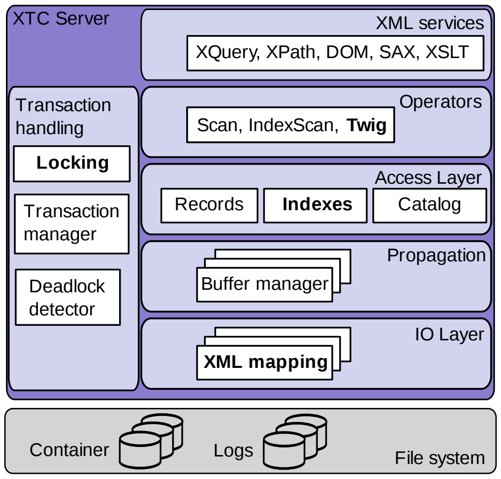

Fig. 1. XTC Architecture – overview [15]

While application-specific benchmarks such as XMark or TPoX are designed to pinpoint the essentials of a typical application of a given domain and only use the required resources and functions of it, benchmarks checking layer-specific functionality or common DBMS components have to strive for generality and have to serve for the determination of generic and sufficiently broad solutions. For the on-going system development of XDBMSs, we want to emphasize that benchmarks have to address each layer separately to deliver helpful hints for performance improvement. Local effects (e.g., expensive XML mapping, compression penalties, quality of operator selection among existing alternatives, buffer effects) are often invisible at the XQuery layer and must be analyzed via native

Table 1. Selected XML documents considered

Document	Decription	Size	Depth		Nodes	Paths
			max	avg		
dblp	Computer science index	330 MB	6	3.4	17 Mio	153
unirot	Universal protein resource	1.8 GB	6	4.5	135 Mio	121
lineitem	TPC-H data	32 MB	4	3.0	2 Mio	17
treebnk	Wall street journal records	86 MB	37	8.44	3.8 Mio	220k
SigRec	Sigmod records	0.5 MB	7	5.7	23.000	7
XMark	Artifical auction data	12 MB	13	5.5	324.271	439
		112 MB	13	5.6	3.2 Mio	451
account	TPoX benchmark documents for accounts, orders, and securities	∼ 6 KB	8	4.7	∼ 320	∼ 100
order		∼ 2 KB	5	2.6	∼ 81	∼ 83
security		∼ 6 KB	6	3.5	∼ 100	∼ 53

interfaces or tailored benchmarks. In the following, we show what kind of approaches help to systematically explore performance-critical aspects within the layers of an XDBMS.

4 Storage Mapping

As the foundation of "external" XDBMS processing, special attention should be paid to storage mapping. Native XDBMSs avoid *shredding* to overcome shortcomings in scalability and flexibility [35] and developed tree-like storage structures [15, 18, 34]. One of the most important aspects is stable *node labeling* to allow for fast node addressing and efficient IUD[1] operations on nodes or subtrees. Two classes of labeling schemes emerged where the *prefix-based* schemes proved to be superior to the *range-based* schemes; they are more versatile, stable, and efficient [7, 8]. In addition, they enable prefix compression which significantly reduces space consumption and, in turn, IO costs. Furthermore, a node label delivers the label of all ancestors which is an unbeatable advantage in case of hierarchical intention locking [12].

Another important mapping property deals with XML structure representation – in particular, with clever handling of *path redundancy.* A naive approach is to map each XML element, attribute, and text node to a distinct physical entity leading to a high degree of redundancy in case of repetitive XML path instances. Because all implementations use dictionaries to substitute XML tag names (element, attribute) by short numbers (i.e., integer), we do not consider variations of dictionary encodings, but aimed to eliminate the structure part to the extent possible. With the help of an auxiliary data structure called *path synopsis* (a kind of path dictionary without duplicates), it is possible to avoid structural redundancy at all [13]. However, text value redundancy has to be addressed by common compression techniques and can be benchmarked orthogonal to the actual XML mapping.

[1] Insert, Update, Delete

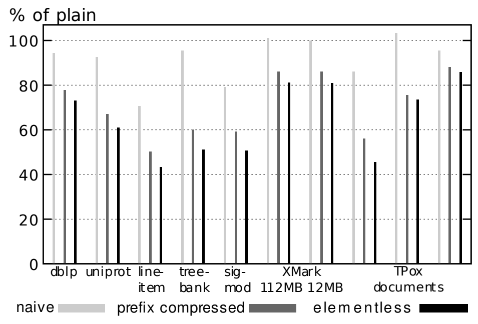

Fig. 2. Mapping space efficiency

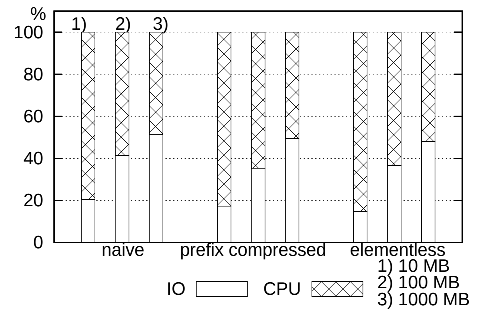

Fig. 3. IO and CPU fractions analyzed

Because ordering is an important aspect for XML processing, the storage mapping has to observe the ordering and to a certain degree the round-trip property. Furthermore, the storage mapping has to be dynamic to allow for IUD operations within single documents and collections of documents. For benchmarking, we identified the following important aspects: space consumption (mapping efficiency), ratio and overhead of optional compression techniques, document size, document no., CPU vs. IO impact, and modifications (IUD).

Benchmark datasets, listed in Table 1, were choosen from different sources to reflect the variety of XML documents. Therefore, we recommend to collect benchmark numbers from real data (e.g., dblp) and artificial data (e.g., XMark), tiny up to huge documents (e.g., TPoX, protein datasets), complex and simple structured documents (e.g., treebank, dblp), as well as document-centric and data-centric XML data (e.g., tpch data, sigmod records). The datasets used are derived from [25, 29] and [33]. For scaling purposes, we generated differently sized XMark documents and differently composed sets of TPoX documents.

Highlighting the fundamental importance of mapping efficiency, Fig. 2 gives some basic insight into three different XML mapping approaches, namely *naive*, *prefix compressed*, and *elementless*. While the *naive* approach simply maps each XML entity combined with its node label to a physical record, the *prefix compressed* mode drastically reduces this space overhead through prefix compression of node labels. The final optimization is to avoid redundancy in structure part at all, by the so-called structure virtualization where all structure nodes and paths can be computed on demand by using the document's *path synopsis*. The mapping efficiency analysis in Fig. 2 shows that all kind of documents benefit from optimized mappings when compared to *plain* (the external XML file size).

The second benchmark example (see Fig. 3) identifies the impact of the mapping's compression techniques. For this purpose, we used TPoX document sets in the scaling range from 10 MB to 1000 MB and analyzed the share of IO and CPU time spend to randomly access and process these sets. For small sets ($\leq$ 100 MB), the reduction of the IO impact is visible, whereas for large sets ($>$ 100 MB) the differences are nearly leveled out. However, several other

Table 2. Characteristics for selected XMark indexes build for an elementless document

#	Type	Definition	Size	Paths	Clustering	Entries
I_1	CAS	//* *(all text nodes)*	25.9 MB	514 (94 %)	label	1,173,733
I_2	CAS	//* *(all text nodes)*	25.9 MB	514 (94 %)	path	1,173,733
I_3	CAS	//item/location	0.26 MB	6 (1.1 %)	label	21,750
I_4	CAS	//asia/item/location	0.025 MB	1 (0.2 %)	label	2,000
I_5	PATH	//keyword	0.67 MB	99 (18 %)	label	69,969
I_6	PATH	//keyword	0.43 MB	99 (18 %)	path	69,969
I_7	CONTENT	*all content nodes*	21.3 MB	-	-	1,555,603
I_8	ELEMENT	*all element nodes*	10.2 MB	-	-	1,666,384
I_9	CAS	//* $\wedge$ //@	31.0 MB	548 (100 %)	path	1,555,603
D_1	DOCUMENT	*document index*	94.5 MB	-	-	1,568,362

aspects heavily influence a storage-related benchmark, e.g., the block size chosen, the available hardware (disk, CPU(s), memory), and the software (OS, load).

Summarizing, the benchmarks have exhibited the influence of various mapping options and available performance spectrum, even at the lower system layers. Because these mappings serve different objectives, it depends on the specific XML document usage, which option meets the actual workload best.

5 Indexes

Although most XML indexing techniques proposed are developed to support reader transactions, they are necessarily used in read/write applications. After all, they have to observe space restrictions in practical applications. Therefore, a universal index implementation has to meet additional requirements such as maintenance costs and footprint. Moreover, its applicability, scalability, and query support have to be analyzed, too.

On the one hand, we can find elementary XML indexes such as DataGuide [9] variations that index all elements of an XML document or full-text indexes covering all content nodes. Such indexes cover a fairly broad spectrum of search support, but need a lot of space and induce high maintenance costs. On the other hand, we can find more adjusted indexes such as path indexes [9, 24] or CAS indexes [20] to fill existing gaps for specific access requirements.

Comparing different index types should overcome common pitfalls, i.e., specialized indexes should not be evaluated against generic indexes, because different sets of indexed elements lead to different index sizes, selectivities, and, therefore, expressive power. Another drawback may be induced by unfavorable or complex update algorithms leading to the difficult question which kind of workload is supported best. Moreover, destroying and rebuilding of indexes may be frequently required on demand in highly dynamic environments, thereby drawing the attention primarily to index building costs. Eventually, query evaluation may be affected by costly index matching, in contrast to the relational case where an attribute-wise index matching is fairly cheap.

Thus, to benchmark index configurations in sufficient quality, the performance of an abundant range of documents, workloads, and index definitions should be evaluated under the different storage mappings. However, due to space restrictions, we can only refer to a set of examples based on *elementless* storage to identify which (of the many) aspects must be observed first when benchmarking indexes. Therefore, Table 2 shows a selection of different index types and their characteristics supporting different kind of queries.[2] For instance, the CAS indexes I_1 and I_2 are equal except for their clustering techniques used, which either optimize document-ordered access or path-based access. Moreover, the path-based clustering may need an additional sort to combine entries from more than one path instance. The indexes I_3 and I_4 serve as examples for refinement; the more focused an index definition, the less XML entities are addressed, which leads to smaller (and in case of IUD to cheaper maintenance of) indexes. However, their expressive power and usability for query support is reduced by such refinements. Path indexes (e.g., I_5 and I_5) using prefix compression on their keys may differ in size, in contrast to CAS indexes where the index size is independent of the clustering. Furthermore, they can exploit optional clustering whose performance benefit is, however, query dependent. Storage-type-independent indexes such as the stated content index I_7 and element index I_8 are fairly generic by covering the entire XML document. Thus, they need maintenance for each IUD operation, but often provide limited fallback access and can thereby avoid a document scan.

Moreover, the complexity of XML indexing is shown by I_7 and I_9 which actually index the same nodes (text and attribute content). But I_9 needs more space to include path information and supports clustering. Therefore, XML indexes require fine-tuning to exploit their features and have to be tailored to the workload. In contrast, unknown workloads may benefit from more generic index approaches, whereas fixed workloads may be best supported by very specific path indexes or CAS indexes.

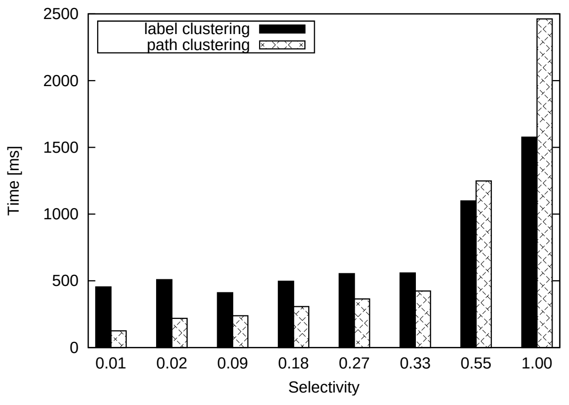

Fig. 4. Path index cluster comparison

Secondary features like the clustering may have a huge impact on query performance. This is confirmed by the indexing example in Fig. 4, which clearly shows that such details need to be considered for XML index benchmarking.

[2] All indexes are built for an 112 MB XMark document containing a subset of the document's 3,221,913 XML entities (element, attribute, and text nodes).

Table 3. Tree-pattern queries used to benchmark join algorithms on XMark documents

#	Query	Matches
X1	/site//open_auction[.//bidder/personref]//reserve	146982
X2	//people//person[.//address/zipcode]/profile/education	15857
X3	//item[location]/description//keyword	136260
X4	//item[location][.//mailbox/mail//emph]/description//keyword	86568
X5	//item[location][quantity][//keyword]/name	207639
X6	//people//person[.//address/zipcode][id]/profile[.//age]/education	7991

Comparing the cluster impact of path indexes I_5 and I_6, it becomes obvious that low selectivities (i.e., small numbers of path classes) are better supported by path-based clustering, whereas high selectivities (in our example $\geq 50\,\%$) can better take advantage of document-ordered, label-based clustering.

6 Path-Processing Operators

With each level of abstraction in the system architecture, the objects become more complex, allowing more powerful operations and being constrained by a larger number of integrity rules. Therefore, the parameter space of the operators frequently increases dramatically such that exhaustive analysis is not possible anymore. Because the options of the data structures and related operations at the path-processing layer are already so abundant and offer so many choices that it becomes hopeless to strive for complete coverage. Nevertheless, accurate-enough benchmarking needs to consider the most influential parameters (e.g., stack size(s), index usage, recursion, false positive filtering) at least in principle.

Here, we sketch our search for optimal evaluation support concerning tree-pattern queries and how we coped with their inherent variety and complexity. Because so many path-processing operators and join mechanisms were proposed in the literature for the processing of query tree patterns (QTP) and because we wanted to check them with our own optimization ideas, we implemented for each of the various solution classes the best-rated algorithm in XTC to provide an identical runtime environment and to use a full-fledged XDBMS (with appropriate indexes available) for accurate cross-comparisons: Structural Joins, TwigStack, TJFast, Twig2Stack, and TwigList [2, 6, 17, 19, 22]. Structural Join as the oldest method decomposes a QTP into its binary relationships and executes them separately. Its key drawback is the high amount of intermediate results produced during the matching process. TwigStack as a holistic method processes a QTP as a whole in two phases, where at first partial results for each QTP leg are derived, before the final result is created in an expensive merging phase. TJFast, inspired by TwigStack, aims at improvements by reducing IO. It uses a kind of prefix-based node labeling which enables the mapping of node labels to their related paths in the document. As a consequence, only document nodes potentially qualifying for QTP expressions have to be fetched, but it is still burdened by the expensive merging phase. Twig2Stack and its refined version

TwigList evaluate QTPs without merging in a single phase, but they require more memory than TwigStack and TJFast. In the worst case, they have to load the entire document into memory.

We complemented the set of these competitors with tailored solutions – especially developed in the XTC context to combine prefix-based node labeling and path synopsis use –, called *S3* and its optimized version *OS3* [17], where query evaluation avoids document access to the extent possible.

To figure out the query evaluation performance for them, we used a set of benchmark queries (see Table 3) for XMark documents which guaranteed sufficient coverage of all aspects needed to cross-compare the different path processing and join algorithms under the variation of important parameters (type of index, selectivity of values, evaluation mechanism (bottom-up or top-down), size of documents, etc.).

Unlike all competitor methods, *S3* and *OS3* executed path expressions not directly on the XML document, but first evaluated them against a path-synopsis-like structure, to minimize access to the document. Hence, variations of our idea underlying the *S3* algorithm outperformed any kind of conventional path operator use, achieved stable performance gains and proved their superiority under different benchmarks and in scalability experiments [17].

Fig. 5 shows our experimental results for the XMark (scale 5) dataset. Because the execution times for Structural Joins were typically orders of magnitude worse than those of the remaining methods, we have dropped them from our cross-comparison. As a general impression, our own methods – in particular, *OS3* – are definitely superior to the competitors, in terms of execution time and IO time. As depicted in Fig. 5 (a), *OS3* is at least three times faster than the other methods. *S3* also obtains the same performance except for *X3*, *X4*, and *X5*. Here, *S3* is about three times slower than *TJFast* for *X4* and *X5* and it is 1.3 times slower for *X3*; here, it exhibits the worst performance among all methods. As a result, processing time and IO cost for queries like *X4* (see Fig. 5 (a) and (b)) are very high; *OS3* can reduce these costs by tailored mechanisms [17]. As a consequence, *OS3* is often more than two times faster than *TJFast* – the best of the competitor methods – for *X3*, *X4*, and *X5*.

Eventually such twig operators should be confronted with largely varying input sizes to prove their general applicability, because stack-based operators (e.g., TwigStack, Twig2Stack) or recursive algorithms stress memory capabilities more than iterative algorithms and iterator-based operators. Another aspect, not addressed in this work, is the preservation of document order for XML query processing. A fair evaluation has to ensure that all operators deliver the same set and order of results. Moreover, a comparison has to state if indexes were used and if the competitive operators used different indexes.

7 Transaction Processing

Like all other types of DBMSs, also XDBMSs must be designed to scale in multi-user environments with both queries and updates from concurrent clients.

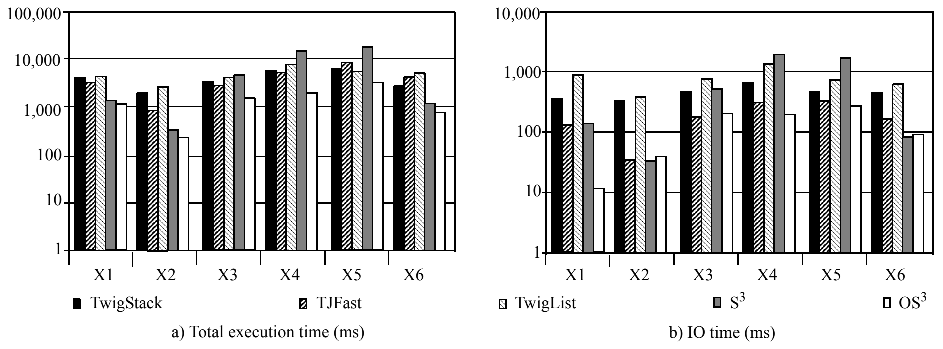

Fig. 5. Experimental results for tree-pattern queries on XMark (scale 5)

Of course, we should here leverage experience collected in decades of database research, but we must also revise appropriateness of prevailing principles. As already mentioned, aspects of logging and recovery are generally not different for transactional XML processing, because all current storage mappings are still build on page-based data structures. In terms of transaction isolation, however, we have to meet concerns of XML's hierarchical structures and new data access patterns.

The TPoX benchmark – the first XML benchmark that covers updates – defines a workload mix that queries and updates large collections of small documents. The authors claim that this setup is typical for most data-centric XML applications, which implies that relevant documents are easily identified through unique attribute values supported by additional indexes[3]. Hence, document-level isolation would always provide sufficient concurrency. In general, however, data contention increases rapidly with the share of non-exact queries like, e.g., range-queries, and the ratio between document size and number of documents.

Research-focused transaction benchmarks for XDBMS should take this aspects into account, should not restrict themselves solely to current XML use cases, and, thus, close the door for new types of applications profiting from the use of semi-structured data. Therefore, we strive for an application-independent isolation concept, which provides us with both competitive performance through simple document-level isolation if sufficient and superior concurrency for fine-grained node-level isolation if beneficial.

The essence of our efforts is a hierarchical lock protocol called taDOM [16]. It bases on the concepts of multi-granularity locking, which are used in most relational DBMSs, but is tailored to maximize concurrent access to XML document trees. To schedule read and write access for specific nodes, siblings, or whole subtrees at arbitrary document levels, transactions may choose from sophisticated lock modes that have to be acquired in conjunction with so-called intention locks on the ancestor path from root to leaf. The concept of edge

[3] 90% of the TPoX workload directly addresses relevant document(s) through unique id attribute values supported by additional indexes for the measurements.

locks [16] complements this approach to avoid phantoms during navigation in the document tree.

To exemplify the benefits of node-level locking in terms of throughput and scalability, we executed a mix of eight read-only and update transaction types on a single 8 MB XML document and varied the number of concurrent clients. All transaction types follow a typical query access pattern. They choose one or more jump-in nodes directed by a secondary index and navigate from there in the document tree to build the query result or find the update position (see also [3]). The results in Fig. 6 show that the reduced locking overhead of document-level isolation only paid off in the single client case with a slightly higher throughput. In all other cases, node-level isolation improved not only the transaction rates, but also the scalability of the system, because a too coarse-grained isolation dramatically increases the danger of the so-called convoy effect. It arises if a system cannot scale with the rate of incoming client requests, because the requested data is exclusively accessed by update transactions. Accordingly, the more clients we had, the more document-level isolation suffered from rapidly growing request queues leading consequently to increasing response times, more timeouts, and finally a complete collapse.

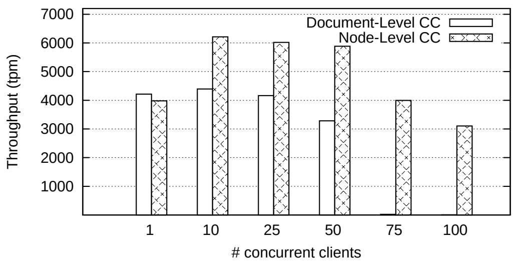

Fig. 6. Scalability of node-level locking

Although this example nicely illustrates the potential of node-level locking, it creates new challenges that we have to conquer. One of the first experiments with fine-grained locking, for example, surprisingly revealed that isolation levels lower than *repeatable* achieve less throughput, which is completely different from what we know from relational databases [14]. The reason for this are exploding numbers of request-release cycles for the intention locks on the document tree; a phenomena that was not known from the small-granule hierarchies in relational systems. Further work led to the cognition that prefix-based node labels were not only a sake for storage and query purposes but also for cheap derivation of ancestor node labels for intention locks. Finally, latest results [3] proved that we can overrule the objections that taDOM is too expensive for larger documents. A simple yet effective lock escalation mechanism allows us to balance lock overhead and concurrency benefits dynamically at runtime.

In the latter experiments, we also identified the importance of update-aware query planning as a new research topic. By now, we are not aware of any work that covers the implications of concurrent document access and modification and the danger of deadlocks during plan generation. Another open question is the concurrency-aware use of the various XML indexes and their interplay with document storage.

8 Integrated View

In addition to component-level, respectively, layer-level analysis, we must also take the dependencies and implications of certain design decisions into account. As already indicated in the previous sections, key aspects like, e.g., the chosen node labeling scheme have a huge effect on the whole system. As another example, the virtualization of the inner document structure leverages not only the storage mapping, but also indexing and path-processing operators in higher system layers. On the other hand, however, improvements in one part of the system can also impose new challenges on other system components. In the concurrency experiments in [3], for example, we reached such a high concurrency at the XML level that our storage structures became the concurrency bottleneck – a completely new challenge for XML storage mappings.

Obviously, the next step towards scalable and generic XDBMS architectures must turn the attention to the interplay of all layers at the system boundary. For the beginning, we can fall back on existing toolboxes to evaluate the performance of XML key functions like navigation (DOM), streaming (SAX, StaX), and path evaluation (XPath). Thereafter, a wide range of XQuery benchmarks should be applied to identify the pros and cons of an architecture. Of course, it seems not possible to cover all potential performance-relevant aspects in every combination with exhaustive benchmarks, but, based on our experience, we can say that we need meaningful combinations of the following orthogonal aspects to get a thorough picture. The database size should be scaled in both directions, the number of documents and the individual document size. To cover the full range of XML's flexibility, different degrees of structure complexity should be addressed with unstructured, semi-structured, structured, and mixed-structured documents in terms of repeating "patterns". Furthermore, variations of parameters such as fan-out and depth can also give valuable insights. Queries should assess capabilities for full-text search, point and range queries over text content, as well as structural relationships like paths and twigs and combinations of both. Update capabilities should be addressed by scaling from read-only workloads over full document insertions and deletions to fine-grained intra-document updates. Finally, these workloads have to be evaluated in single and multi-user scenarios.

To identify meaningful combinations, there must be an exchange between database researchers and application developers. On the one hand, specific application needs must be satisfied by the XDBMS and, on the other hand, applications have to be adjusted to observe the strengths and weaknesses of XDBMSs. Although the flexibility of XML allows many ways to model data and relationships in logically equivalent variants, it may have a strong influence on the performance of an XDBMS, e.g., in terms of buffer locality. Hence, system capabilities will also cause a rethinking the way how to model XML data, because data modeling driven solely by business needs will not necessarily lead to an optimal representation for an XDBMS-based application. Consequently, there must be a distinction between logical and physical data modeling as in the relational world.

9 Conclusions

We believe that benchmarking is a serious task for database development and, furthermore, we think that current benchmarks in the XML domain do not cover the entire XML complexity provided by native XDBMSs. To reveal important insights how database-based XML processing (e.g., XQuery) is executed, we started to implement various and promising algorithms for all layers in the entire DBMS architecture. Although this approach is time-consuming (and sometimes error-prone, too), it allows for direct comparisons and analyses of competitive ideas and justifies the development of our own native XDBMS – XTC. However, in this work we want to motivate the bottom-up development and simultaneous benchmarking, by giving some insight into critical aspects which arose during the development. Furthermore, we emphasize common pitfalls and results gained through tailored benchmarking of distinct components.

In addition, it turned out that benchmarking is an interplay of *hardware*, *software*, *workload* (data and queries), *measuring setup*, and *fairness*. Hardware selection has to be reasonable w.r.t. main memory, number of CPUs, disk size and speed, etc. For instance, the trade-off between CPU and IO costs can either be adjusted through the selection of algorithms or often by hardware adjustments. New concepts such as distributed processing or adaptivity can rapidly extend the benchmark matrix. Thus, benchmarking different algorithms needs to be performed under realistic system configurations (e.g., current and proper-sized hardware). When it comes to workload modeling, either real-world datasets and queries and/or artificial datasets representing a wide range of applications should be used. Unfortunately, this is a difficult problem and needs sound considerations. Moreover, the situation that one algorithm put to benchmark is totally dominating its competitors is rather rare, in fact, most often its preferences are emphasized and shortcomings omitted in scientific contributions.

New findings by benchmarking gained new techniques and alternative algorithms may lead to a rethinking, and thereby to a reimplementation, which may trigger expensive development costs, too. Thus unfortunately, the integration of new ideas is slow and cumbersome. Here, research is challenged to evaluate by proof-of-concept implementations such new ideas, before commercial systems may adopt them.

9.1 Should We Propose Another XML Benchmark?

A logic conclusion, drawn after evaluating critical aspects of XML processing mentioned in this work and existing XML benchmarks, is to develop another (new) XML benchmark. However, we do not think that a new benchmark is necessary at all. The rich variety of XML workloads (i.e., datasets and queries) allows for generating critical (and corner) cases. For instance, during our storage layer benchmarking we started with single large documents from [25], before we learned that real applications may also need to process several million (small) documents [29]. Therefore, we extended our storage benchmarks to meet all kinds of XML documents. Furthermore, XML processing pervades certain areas such as information retrieval, where fast reads are mandatory, or the area of application logging, where prevalently inserts and updates occur concurrently. Thus, the spectrum of transactional processing is quite wide and requires tailored protocols to ensure ACID capabilities. Thus, to evaluate concurrent transactional behavior, for instance, is possible by weighting pre-defined benchmark queries of [29, 33] according to the objectives put to benchmark.

However, an open system design is helpful to adjust the measuring points for meaningful results. That means, either internal behavior (algorithms) have to be published and implemented into a single system or at least proper interfaces are available on each system put to test.

9.2 Future Work

For our future work, we plan to extend our benchmark findings and continue the bottom-up approach towards the query and application level. Here, we want to address XPath/XQuery in more detail, schema processing, XML applications and use cases, XML data modeling, and the domain of information retrieval on XML. Furthermore, aspects like query translation, query optimization, and XQuery language coverage are critical points for comparing XQuery compilers.

References

1. Afanasiev, L., Franceschet, M., and Marx, M.: XCheck: a platform for benchmarking XQuery engines. In Proc. VLDB (2006) 1247-1250
2. Al-Khalifa et al: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. In Proc. ICDE (2002) 141-152
3. Bächle, S., and Härder, T., and Haustein, M.: Implementing and Optimizing Fine-Granular Lock Management for XML Document Trees. In Proc. DASFAA (2009)
4. Bourret, R.: XML Database Products. http://www.rpbourret.com/xml/XMLDatabaseProds.htm
5. Böhme, T. and Rahm, E.: XMach-1: A Benchmark for XML Data Management. BTW (2001) 264-273
6. Bruno, N., Koudas, N., and Srivastava, D.: Holistic Twig Joins: Optimal XML Pattern Matching. In Proc. SIGMOD (2002) 310-321
7. Christophides V., Plexousakis D., Scholl M., and Tourtounis S.: On Labeling Schemes for the Semantic Web. In Proc. Int. WWW Conf. (2003) 544-555
8. Cohen, E., Kaplan, H., and Milo, T.: Labeling Dynamic XML Trees. In Proc. PODS (2002) 271-281
9. Goldman, R. and Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Proc. VLDB (1997) 436-445
10. Grust, T., Rittinger, J., and Teubner, J.: Why off-the-shelf RDBMSs are better at XPath than you might expect. In Proc. SIGMOD (2007) 949-958
11. Halverson, A., Josifovski, V., Lohman, G., Pirahesh, H., and Mörschel, M.: ROX: relational over XML. In Proc. VLDB **30** (2004) 264-275
12. Härder, T., Haustein, M., Mathis, C., and Wagner, M.: Node Labeling Schemes for Dynamic XML Documents Reconsidered. DKE **60** (2007) 126-149
13. Härder, T., Mathis, C., and Schmidt, K.: Comparison of Complete and Elementless Native Storage of XML Documents. In Proc. IDEAS (2007)

14. Haustein, M., and Härder, T.: Adjustable Transaction Isolation in XML Database Management Systems. In Proc. XSym (2004) 173-188
15. Haustein, M. P., Härder T.: An efficient infrastructure for native transactional XML processing. DKE **61** (2007) 500-523
16. Haustein, M., and Härder, T.: Optimizing lock protocols for native XML processing. DKE **65** (2008) 147-173
17. Izadi, K., Härder, T., and Haghjoo, M.: S3: Evaluation of Tree-Pattern Queries Supported by Structural Summaries, DKE **68** (2009) 126-145
18. Jagadish, H. V. et al: TIMBER: A native XML database. VLDB Journal **11** (2002) 274-291
19. Jiang, H., Wang, W., Lu, H., and Xu Yu, J.: Holistic Twig Joins on Indexed XML Documents. In Proc. VLDB (2003) 273-284
20. Li, H.-G., Aghili, S. A., Agrawal, D., El Abbadi, A.: FLUX: Content and Structure Matching of XPath Queries with Range Predicates. In Proc. XSym (2006) 61-76
21. Lu, H. et al: What makes the differences: benchmarking XML database implementations. ACM Trans. Internet Technol. **5**, (2005) 154-194
22. Lu. J., Ling, T. W., Chan, C. Y., and Chen, T.: From region encoding to extended Dewey: on efficient processing of XML twig pattern matching. In Proc. VLDB (2005) 193-204
23. Michiels, P., Manolescu, I., and Miachon, C.: Toward microbenchmarking XQuery. Inf. Syst. **33** (2008) 182-202
24. Milo, T., Suciu, D.: Index Structures for Path Expressions. In Proc. ICDT (1999) 277-295
25. Miklau, G. XML Data Repository, www.cs.washington.edu/research/xmldatasets
26. Nambiar, U., Lee, M.L., and Li, Y.: Xml benchmarks put to the test. In Proc. IIWAS (2001)
27. Nicola, M., John, J.: XML parsing: a threat to database performance. In Proc. CIKM (2003) 175-178
28. Nicola, M., van der Linden, B.: Native XML support in DB2 universal database. In Proc. VLDB (2005) 1164-1174
29. Nicola, M., Kogan, I., and Schiefer, B.: An XML transaction processing benchmark. In Proc. SIGMOD (2007) 937-948
30. Oracle XML DB 11g www.oracle.com/technology/tech/xml/xmldb/
31. Phan, B. V., Pardede, E.: Towards the Development of XML Benchmark for XML Updates. In Proc. ITNG (2008) 500-505
32. Schmidt, A. et al: Why and how to benchmark XML databases. SIGMOD Rec. **30** (2001) 27-32
33. Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I., and Busse, R.: XMark: a benchmark for XML data management. In Proc. VLDB (2002) 974-985
34. Schöning, D. H.: Tamino - A DBMS Designed for XML. In Proc. ICDE (2001)
35. Shanmugasundaram, J. et al: Relational Databases for Querying XML Documents: Limitations and Opportunities. In Proc. (1999) 302-314
36. XML Path Language (XPath) 2.0. W3C Recommendation. http://www.w3.org/TR/xpath
37. XQuery 1.0: An XML Query Language. W3C Recommendation. (2005) http://www.w3.org/TR/xquery
38. XQuery Update Facility 1.0. W3C Candidate Recommendation. (2008) http://www.w3.org/TR/xquery-update-10/
39. Yao, B. B., Özsu, M. T., and Khandelwal, N.: XBench Benchmark and Performance Testing of XML DBMSs. In Proc. ICDE (2004) 621