Essential Performance Drivers in Native XML DBMSs

Theo Härder, Christian Mathis, Sebastian Bächle,
Karsten Schmidt, and Andreas M. Weiner

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

Abstract. As a multi-layered XML database management system, we have designed, implemented, and optimized over the recent five years our prototype system XTC, a native XDBMS providing multi-lingual query interfaces (XQuery, XPath, DOM). In particular in higher system layers, we have compared competing concepts and iteratively found salient solutions which drastically improved the overall XDBMS performance. XML query processing is critically affected by the smooth interplay of concepts and methods on all architectural layers: node labeling and mapping options for storage structures; availability of suitable index mechanisms; provision of a spectrum of path processing operators; query language compilation and optimization. Furthermore, effective and efficient locking protocols must be present to guarantee the ACID properties for XML processing and to achieve high transaction throughput.

In this survey, we outline our experiences gained during the implementation and optimization of XTC. We figure out the "key drivers" to maximize throughput while keeping the response times at an acceptable level. Because we have implemented all options and alternatives in XTC, dedicated benchmark runs allow for comparisons in identical environments and illustrate the benefit of all implementation decisions[1].

1 Motivation

In recent years, XML's standardization and, in particular, its flexibility (e. g., data mapping, cardinality variations, optional or non-existing structures, etc.) evolved as driving factors to attract demanding write/read applications, to enable heterogeneous data stores, and to facilitate data integration. Because business models in practically every industry use large and evolving sets of sparsely populated attributes, XML is more and more adopted by those companies which have even now launched consortia to develop XML schemas adjusted to their particular data modeling needs. As an example, world-leading financial companies defined more than a dozen XML schemata and vocabularies to standardize data

[1] This work has been partially supported by the German Research Foundation (DFG) and the Rheinland-Pfalz cluster of excellence "Center of Mathematical and Computational Modelling", Germany (see www.cmcm.de).

J. van Leeuwen et al. (Eds.): SOFSEM 2010, LNCS 5901, pp. 29–46, 2010.

processing and to leverage cooperation and data exchange [43]. For these reasons, XML databases currently get more and more momentum if data flexibility in various forms is a key requirement of the application and they are, therefore, frequently used in collaborative or even competitive environments [26].

Native XML database systems (XDBMSs) promise tailored XML processing, but most of the systems published in the DB literature are primarily designed for efficient document storage and retrieval [22,39]. Furthermore, they are optimized to evaluate complex XQuery statements on large XML documents in single-user mode. Hence, many aspects of proven DBMS functionality and technology are often neglected in these systems, in the first place read/write transaction processing in multi-user mode, but also storage and indexing of dynamic XML documents in flexible formats to best satisfy the needs of specific applications.

As a consequence of the growing demand and the increasing adoption of XDBMSs, enhanced functionality and flexibility is needed in all system layers. At the bottom-most layer of the XDBMS architecture, a spectrum of storage devices, e. g., flash disks and magnetic disks, should be supported to provide for high-performance requirements. In upper layers, tailor-made and automatically chosen storage and index structures should help to approach application-specific needs [38]. These structures should enable complex path processing operations which, in turn, have to be integrated into cost-optimized query plans.

Of course, the original "retrieval-only" focus of XDBMSs – probably caused by the first proposals of XQuery respectively XPath where the update part was left out – is not enough anymore. Due to the growing need of update facilities, XDBMSs should efficiently support fine-grained, concurrent, and transaction-safe document modifications. For example, workloads for *financial application logging*[2] include 10M to 20M inserts in a 24-hour day, with about 500 peak inserts/sec. Because at least a hundred users need to concurrently read the data for troubleshooting and auditing tasks, concurrency control is challenged to provide short-enough response times for interactive operations [26]. Currently, all vendors of XML(-enabled) DBMSs support updates only at document granularity and, thus, cannot manage highly dynamic XML documents, let alone achieve such performance goals. Hence, new concurrency control protocols together with efficient implementations are needed to meet these emerging challenges.

During the last five years, we have addressed – by designing, implementing, analyzing, optimizing, and adjusting an XDBMS prototype system called XTC (XML Transactional Coordinator) – all these issues indispensable for a full-fledged DBMS. To guarantee broad acceptance for our research, we strive for a *general solution* that is even applicable for a spectrum of XML language models (e. g., XPath, XQuery, SAX, or DOM) in a multi-lingual XDBMS environment. In this survey paper, we want to report on our experiences gained and, in particular, focus on the concepts, functionalities, and mechanisms which turned out to be essential performance drivers of XDBMSs.

[2] Another example is monitoring the airline traffic control where legal demands call for collecting and saving huge and rapidly growing volumes of heterogeneous information (formatted data, mail, voice, signal, etc.) for 5 years.

2 Hierarchical DBMS Architecture

As mapping model or reference architecture for relational DBMSs, Andreas Reuter and the first author proposed a hierarchical multi-layer model about 25 years ago [17]. The five layers describe the major steps of dynamic abstraction from the physical storage up to the user interface. At the bottom, the database consists of huge volumes of persistently stored bits interpreted by the DBMS into meaningful information on which the user can operate. With each abstraction level, the objects become more complex, allowing more powerful operations and being constrained by a growing number of integrity rules. The uppermost interface supports a data model using set-oriented and declarative operations.

A key observation made while implementing this model in various projects was that the *invariants in database management determine the mapping steps of the supporting architecture*[13]. Hence, for XML database management, these basic invariants should still hold true: page-oriented mapping to external storage, management of record-oriented data, set-oriented database processing. Therefore, we used the five-layer model shown in Fig. 1 as our reference architecture for XTC. Obviously, both lower layers L1 and L2 keep their essential characteristics and functionality as in the relational world, because neither objects (pages or blocks) nor operations (fix, unfix or read/write) change very much. However, much more adaptations are necessary from L3 upwards[3]. In contrast to handling row sets often based on simple TID access or reference, the performance of handling and manipulating sequences of XML subtrees critically depends on a suitable node labeling scheme. In [19], we have already argued that it is *the key to efficient and fine-grained XML processing.*

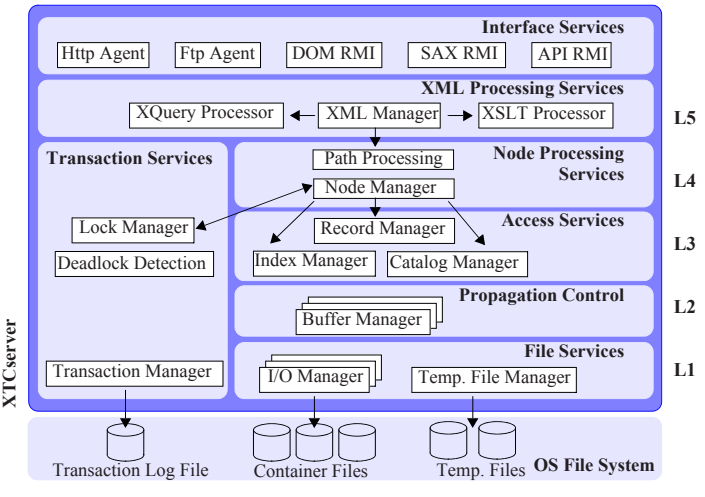


Fig. 1. The five-layer architecture of XTC

[3] For DBMSs, it is especially true: "Performance is not everything, but without performance everything is worth nothing."

3 Node Labeling

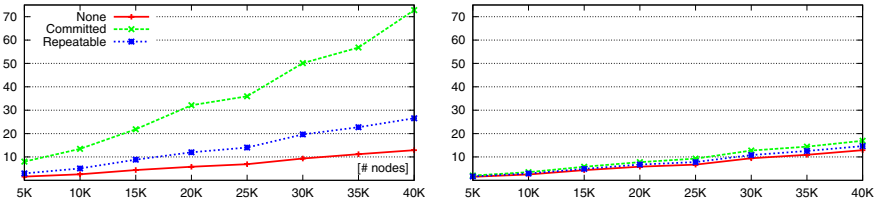
In our first XTC version, we started with a simple, but very inefficient solution by choosing a sequential numbering scheme (SEQIDs) which could only guarantee *uniqueness* and *order preservation* of node labels. Exploring fine-grained XML locking as the initial focus of our research, the protocols frequently had to acquire intention locks on all ancestors of the context node *cn*. To find their node labels, overly expensive look-ups in the disk-based document were unavoidable.

Various range-based and prefix-based node labeling schemes [7] were considered the prime candidates for XDBMSs, because their labels directly enable *testing of all XPath axes*. A close comparison and evaluation of those schemes included other XDBMS-specific criteria [14]. While range-based schemes failed to guarantee *immutable labels in dynamic XML documents* (under heavy updates/insertions) and could not directly compute, i. e., without further index access or similar deviation, *all ancestor labels* of *cn*, prefix-based node labeling turned out to be the winner, because they support all desired labeling properties *without the need of document access*. Each label based on the Dewey Decimal Classification, e. g., $d_1 = 1.7.9.5.17$, directly represents the path from the document's root to the related node and the local order w. r. t. the parent node. Some schemes such as OrdPaths [34], DeweyIDs, or DLNs [14] provide immutable labels by supporting an overflow technique for dynamically inserted nodes. Because they are equivalent for all XDBMS tasks, we use the generic name *stable path labeling identifiers* (SPLIDs) for them.

Because SPLIDs tend to be space-consuming, suitable encoding and compression of them in DB pages is a must. Effective encoding of SPLID divisions at the bit level may be accomplished using Huffman codes [14]. It is important that the resulting codes preserve their order when compared at the byte level. Otherwise, each comparison, e. g., as keys in B*-trees or entries in reference lists, requires cumbersome and inefficient decoding and inspection of the bit sequences. Because such comparisons occur extremely frequent, schemes violating this principle may encounter severe performance problems [25].

When SPLIDs are stored in document sequence, they lend themselves to prefix-compression and achieve impressive compression ratios. Our experiments using a widely known XML document collection [32] confirmed that prefix-compression reduced the space consumed for dense and non-dense SPLID orders down to ∼15 – ∼35% and ∼25 – ∼40%, respectively [15].

To see the hidden gain of SPLIDs for lock-related costs, we generated a variety of XML documents consisting of 5,000 up to 40,000 individual XML nodes and traversed the documents under various isolation levels [11]. Although we optimized SEQID-based access to node relatives by so-called on-demand indexing, the required lock requests were directly translated into pure lock management overhead as plotted in Fig. 2(a). We have repeated document traversal using SPLID-based lock management (see Fig. 2(b)). Because the difference between *none* and *committed/repeatable* is caused by locking overhead, we see drastic performance gains compared to SEQIDs. While those are responsible for an up to ∼600% increase of the reconstruction times in our experiment, SPLIDs keep



(a) SEQID node labeling (b) SPLID node labeling

Fig. 2. Documents traversal times (sec.)

worst-case locking costs in the range of ∼10 – ∼20% [3]. SEQIDs have fixed length, whereas SPLIDs require handling of variable-length entries. Coping with variable-length fields adds some complexity to SPLID and B*-tree management. Nevertheless, reconstruction time remained stable when SPLIDs were used – even when locking was turned off (case *none*).

Comparison of document reconstruction in Fig. 2(a) and (b) reveals for identical XML operations that the mere use of SPLIDs improved the response times by a factor of up to 5 and more. This observation confirms that prefix-based node labeling is indispensable for internal XML navigation and set-based query processing, but also for the lock manager's flexibility and performance.

4 Storing and Indexing Documents

Based on specific document characteristics, storage management should provide for automatic selection of appropriate mapping formats and adjusted parameters [38]. Typical methods replace the element/attribute names of the *plain* (external) format by *VocIDs* to save space and need some *Admin* metadata to enable variable-length entries. Inner tree nodes, i. e., the "structure", are stored as records containing $< SPLID, VocID, Admin >$, whereas leaf nodes carry the "content" in $< SPLID, Value, Admin >$ records.

4.1 Storage Formats

Three kinds of mappings are provided in XTC. The document-oriented storage formats keep both content and structure: Using the *naive* format, the VocIDs of all element/attribute and content nodes are directly mapped together with their uncompressed SPLIDs to the underlying storage structure (see Fig. 3), whereas the *pc* format deviates from the *naive* mapping by applying prefix-compression to all SPLIDs. As a novel mapping approach, the path-oriented storage format called *po* virtualizes the entire structure part of the document.

For this reason, an auxiliary, document-related structure called *path synopsis* is needed. It represents for each document path its path class and is enhanced by *path class references* (PCRs) for them (see Fig. 4(a)). Because providing substantial mapping flexibility, effective lock management support, and also considerable

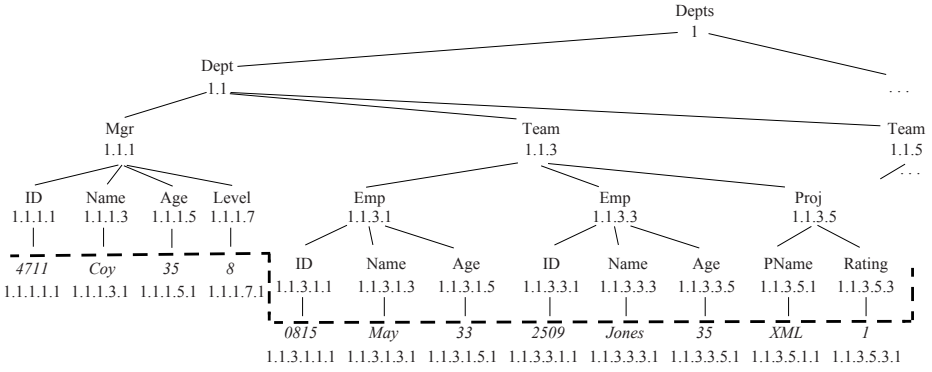


Fig. 3. Document fragment (in the path-oriented storage format, only nodes below the dashed line are physically stored)

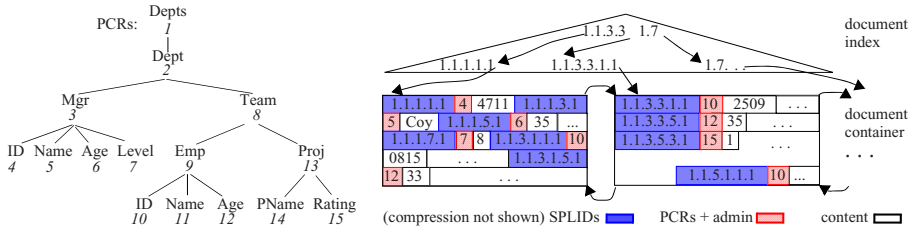
speed-up of query evaluation [15], the use of path synopses turned out to be a *key concept* for XTC's processing efficiency.

Only the "content part" is physically stored when the *po* format is used (see Fig. 3). Reference [31] explains the concept of *structure virtualization*, i. e., the *po* mapping, in detail and shows that path reconstruction can be achieved on demand when the SPLID of a node together with its PCR is present. For this reason, leaf records are composed of $< SPLID, Value, PCR, Admin >$ where the SPLIDs are prefix-compressed. All navigational and set-oriented operations can be executed guaranteeing the same semantics as on *naive* or *pc* formats. Fig. 4(b) shows that only the content nodes are stored; using the path synopsis, entry $< 1.1.1.5.1, 6, 35 >$ tells us that the related path to the value 35 is `/Depts/Dept/Mgr/Age` with the ancestor SPLIDs 1, 1.1, 1.1.1, 1.1.1.5.

All documents are physically represented using a B*-tree as base structure, where the records (tree nodes) are consecutively stored in the *document container* thereby preserving the document order. The *document index* is used to provide direct access via SPLIDs. As an example, Fig. 4(b) illustrates the *po* format for the document fragment of Fig. 3.

As compared to the *plain* format, *naive* as the straightforward internal format typically achieves a storage gain of ∼10% to ∼30%, although the saving from VocID usage is partially compensated by the need for node labels. Extensive empirical (structure-only) tests using our reference document collection [32] have identified a further gain of ∼27% to ∼43% when using *pc* format and, in turn, a *naive*-to-*po* gain of ∼71% to ∼83% [15]. Because also exhibiting better mapping and reconstruction times, the *po* format is a substantial performance driver.

Content compression is orthogonal to the storage formats discussed. We have observed [15] that, using simple character-based compression schemes, the content size could be considerably reduced in our rather data-centric reference document collection such that a storage gain of ∼22% to ∼42% is possible. Even more compression gain could be expected for document-centric XML content.



(a) Path synopsis (b) Stored document in *po* format

Fig. 4. Physical storage structures

4.2 Indexing Options

Set-oriented access to the nodes of an XML document is supported by a variety of index types. Similar to the document store including the document index (see Fig. 4(b)), all secondary index types in XTC are implemented using B-tree/B*-tree structures:

- *Element index*: It offers two basic access primitives: *Scan* and *Axis Evaluation*. For this reason, it maintains for each element name a reference list of all its nodes. All element names are organized in a *name directory* where the reference lists are themselves indexed (*node-reference indexes*).
- *Path index*: This structure can index paths qualified by a simple path predicate p, e. g., `//Mgr/Age` or `//Dept//Emp`. Because SPLIDs carry essential path information, they are utilized together with the path synopsis to directly support path queries.
- *Content index*: It maps each content value to the text nodes which stores it.
- *Content-and-Structure (CAS) index*: As a hybrid index combining content and structure information, it supports the evaluation of CAS queries. Each content value is associated with a list of references (SPLID + PCR) to the related document nodes. Such a combined reference enables together with the path synopsis the reconstruction of the entire path without accessing the document.

CAS indexes are particularly powerful, because a large share of matching queries can be evaluated solely on the index structure. Only when additional attributes/elements are requested for output, access to the disk-based document is needed. In a *unique* CAS index, all entries have the same PCR, while in a *homogeneous collective* index, the entries may have varying PCRs, i. e., they may refer to different path classes. For the *heterogeneous collective* CAS index, the index predicate p may be generalized to $p = p_1 \vee ... \vee p_i \vee ... \vee p_n$ where the p_i are simple path predicates. A *generic* CAS index contains all values of a certain type, e. g., $p =$ `//*` [29].

Refined evaluations of XTC's indexing performance can be found in [31]. Furthermore, it is reflected by the query evaluation results reported in Sect. 7.

5 Path Processing Operators

So far, layer L4 of XTC provides about 50 *path processing operators* (PPOs) – exhibiting locking-aware behavior where appropriate [30] – which are tailored to the underlying storage and index structures (L3). They can be considered as part of the physical algebra operations. Here, we can only focus on a prominent PPO generally called *holistic twig join*. A twig query (also called tree-pattern query) contains multiple path branches (twigs) and potentially path and content predicates, e. g. `doc('dept.xml')//Mgr[./Age>='50']/Name)` as XPath expression. It can be either decomposed into single paths or processed as a whole. Single paths could be evaluated by structural joins or matched by means of indexes and then joined (or intersected). To avoid joins, special (twig) indexes can answer path pattern queries directly. In contrast to a structural join, a holistic twig join can consume more than two input streams which are combined to match the complex branching path patterns.

As identified in Fig. 5, numerous algorithms were proposed for twig processing, but no algorithm obtains the expressiveness of our (logical) twig operator called

Algorithms for Holistic Twig Joins	descendant	child	and	or	not	optional edges	projection	grouping	expressions	filters	pos. predicates	no. of phases	element indexes	path indexes
PathStack [2]	+											-		
PathStack ¬ [19]	+				+							-		
TwigStack [2]	+		+									2	+[1]	
TwigStackList [22]	+	+	+									2		
TwigStackList ¬ [38]	+	+	+		+							2		
TJFast [23]	+	+	+									2		+[2]
iTwigJoin [3]	+	+	+									2		+[3]
TSGeneric + [18]	+		+									2	+[4]	
Twig^{2}Stack [4]	+		+			+	+	+				1		
TwigList [29]	+		+				+					1		
TwigOpt. [7]	+		+	+			+					1	+	
Ext. TwigOpt [25]	+	+[5]	+	+	+[5]	+	+	+	+	+[5]	+	1	+	+[6]

1. Skipping in TwigStack only supported by XB-Tree.
2. TJFast requires special embedding of path information into SPLIDs.
3. iTwigJoin supports streams generated by path indexes, but no internal element reconstruction.
4. TSGeneric + relies on the special XR-tree.
5. Matching *child / not / filter* integrated in output generation (and not in matching phase).
6. Index embedding with *ancestor tuple builder* algorithm only possible, when SPLIDs are indexed.

Fig. 5. Survey of twig algorithms

Extended TwigOpt. We consider this operator richness as desirable, because the higher the expressiveness, the more operations can be embedded into the twig algorithm. Hence, the number of operators can be minimized in the final query execution plan (QEP) (see Sect. 6). Therefore, our twig operator includes so many concepts: path pattern supporting axes *child*, *descendant* and *attribute*; logical *and* and *or* conjunctions; optional subtree patterns (i. e., optional edges); projection; positional predicates; output filters; embedded output expressions; grouping.

Depending on the indexes present, it is physically mapped to suitable storage structures during QEP optimization (see Sect. 6). Again, its potential as a performance driver is revealed in Sect. 7.

6 Query Planning and Optimization

For XQuery translation and optimization, we referred to a QGM-based (query graph model [12]) internal structure guiding the entire process of query planning

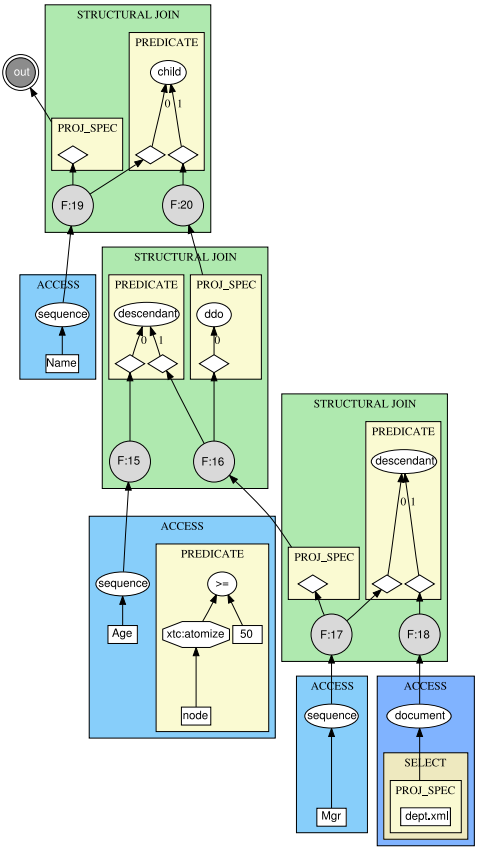


Fig. 6. XQGM instance of the query

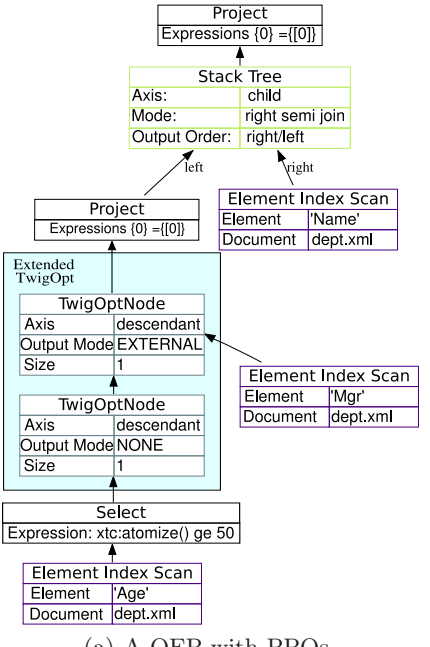
and optimization. We substantially extended this model to XQGM [29] to primarily enable *query decorrelation rewrites* [35], i. e., replacing nesting by joins, and to integrate *index support*, i. e., the mapping of XPath/XQuery expressions to our rich collection of index types. With cardinality estimations derived from the related XML documents [1], our cost model can be tailored to the specific XQuery evaluation [42]. In particular, the optimizer tries to apply the more-than-usual expressiveness and functionality of *Extended TwigOpt* combined with CAS index support to minimize operator use and to unlatch XTC's evaluation power.

To only sketch the idea and to limit the explanation needs, the following simple XQuery statement serves as a query planning and optimization example:

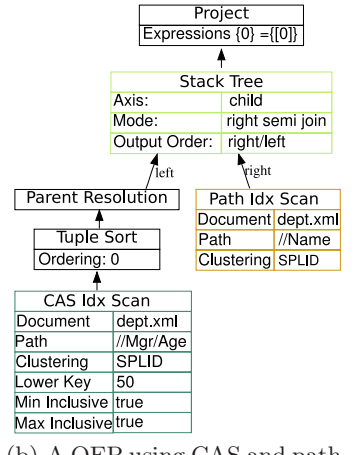
```
for $Dept in doc('dept.xml')//Mgr[./Age>=''50''] return $Dept/Name
```

This query returns the department names of managers that are at least 50 years old. As indicated before, the XQGM suits as our logical XML algebra. Fig. 6 shows the XQGM instance that corresponds to the query. Here, the structural predicates (*child* and *descendant* axes) are evaluated using *structural joins* (SJs). The SJs receive their inputs from access operators that work on nested tuple sequences. SJ inputs are connected to so-called *tuple variables* (F:x) having a for-loop semantics, i. e., an SJ is similar to a relational sort-merge join [29].

For the execution of this query, numerous QEPs can be derived. Due to structural join reordering and various indexing methods that can be considered as



(a) A QEP with PPOs



(b) A QEP using CAS and path indexes

Fig. 7. Optimization alternatives

alternatives by our query optimizer [42], the search space quickly becomes very large. Therefore, cost-based query optimization is necessary to wipe out expensive QEPs. Amongst others, Fig. 7(a) and (b) show two possible QEPs. In Fig. 7(a), the structural predicates are evaluated using the PPOs *StackTree* (Structural Join) and *Extended TwigOpt* (Holistic Twig Join). This QEP is gained by performing query rewrite using join fusion [41]. Both operators receive their inputs by accesses to the element index. Even though, a joint application of *StackTree* and *Extended TwigOpt* can outdistance QEPs that only consist of *StackTree* operators [41], Fig. 7(b) shows a more efficient variant. In this case, we assume a CAS index (`//Mgr/Age [Integer]`) and a path index (`//Name`). Instead of filtering all `Age` nodes and costly evaluating the structural predicates for `//Mgr/Age`, the optimizer exploits both types of indexes and connects their results by a *StackTree* operator. On large documents, this alternative is expected to outperform the former one by several orders of magnitude, because CAS indexes and path indexes are similar to materialized views.

7 Query Evaluation Performance

To sketch the interplay and efficiency of PPOs and query optimization, we want to repeat some results of an empircal study [31] and give some speed-up figures illustrating performance gains of index-supported range queries. We used the XMark framework [37] to evaluate in five different cases (C1 – C5) range query `//Asia/Item/[‘‘C’’ ≤ Location ≤ ‘‘G’’]`, where all tests were carried out on 4 XMark documents of size 10 MB, 50 MB, 100 MB, and 500 MB:

C1: No CAS/content index is available; hence, a holistic twig join operator had to be used.
C2: A content index for all content nodes is present, allowing structure predicate evaluation by the twig join operator. The delivered SPLIDs are not in document order and have to be sorted to serve as input for the twig join.
C3: A generic CAS index (`//*[String]`) enables PCR matching to remove false positives.
C4: A collective CAS index (`//Item/Location [String]`) is more focused than the generic index.
C5: A unique CAS index (`//Asia/Item/Location [String]`) takes care that no false positives can occur.

We refer to case C1 as baseline – no index was present and verification of the content predicate required navigational steps (thus implying expensive random IO) – and illustrate in Fig. 8 that *three orders of magnitude* can be gained by adjusted indexes. While content access support in case C2 achieved some noticeable improvements for the twig join, the real performance boost was observed for cases C3, C4, and C5 exploiting CAS indexes and PCR structure matching such that joins were not needed anymore. Thus, speed-ups in these cases increase with the document sizes by *up to two orders of magnitude*. Note, in cases C1 and C2, missing or insufficient index support caused linear response time growth w.r.t.

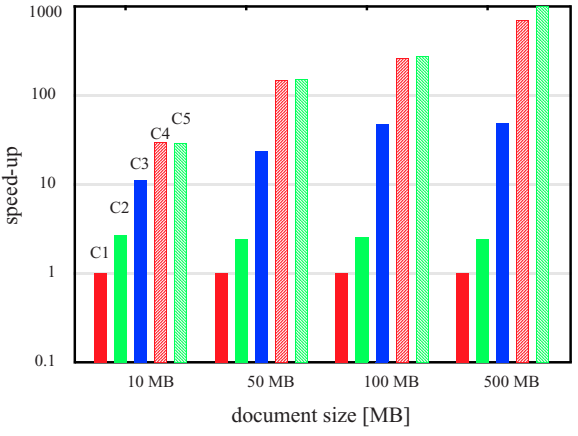


Fig. 8. Index-supported range queries

document sizes, whereas the response times in the remaining cases increased only marginally due to CAS support. This effect enhanced the speed-up factors observed for large document sizes.

Further performance gains and, at the same time, energy savings are possible when flash disks are used. Compared to magnetic disks, these storage devices provide a factor of 100 and more IOPS for random reads (and much lower, but steadily improved random-write performance). Hence, IO-intensive DB applications greatly take advantage of these properties. Initial experiments revealed that XTC in its current version improved its transaction throughput by a factor of ∼3 thereby using less energy [16].

8 XML Locking

Multi-granularity lock (MGL) protocols [10,11] have introduced IR, IX, R, U, and X locks to achieve fine-granularity locking on hierarchies. Always locking entire subtrees, they are too strict for XML transactions because writers can sometimes be tolerated in the subtree of a context node *cn* [21].

8.1 Locking Concepts of taDOM

In the context of XTC, we developed a novel approach to XML concurrency control called *taDOM* providing tailor-made modes for fine-grained XML locking [18]. taDOM renames the conventional MGL locks and introduces new lock modes for *single nodes* called NR (node read) and NX (node exclusive), and for *all siblings under a parent* called LR (level read). The novelty of the NR and LR modes is that they allow, in contrast to MGL, to read-lock only a node or all nodes at a level (under the same parent), but not the corresponding subtrees.

To enable transactions to traverse paths in a tree having (levels of) nodes already read-locked by other transactions and to modify subtrees of such nodes,

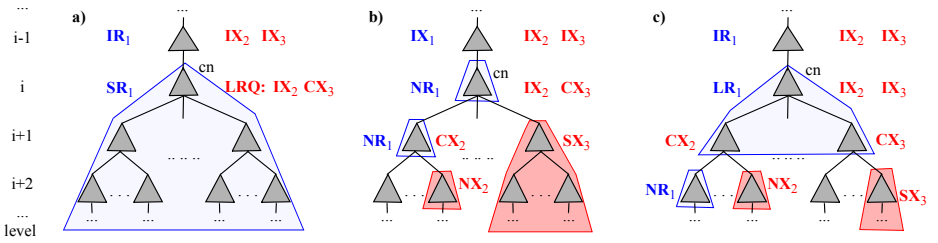


Fig. 9. Examples of locking flexibility and effectivity using taDOM's concepts

a new intention mode CX (child exclusive) had to be defined for a context (parent) node. It indicates the existence of an SX or NX lock on some direct child nodes and prohibits inconsistent locking states by preventing LR and SR locks. It does not prohibit other CX locks on a context node c, because separate child nodes of c may be exclusively locked by other transactions (compatibility is then decided on the child nodes themselves). Altogether these new lock modes enable serializable transaction schedules with read operations on inner tree nodes, while concurrent updates may occur in their subtrees. An important and unique feature (not applicable in MGL or other protocols) is the optional variation of the *lock depth* which can be dynamically controlled by a parameter. Lock depth n determines that, while navigating through the document, individual locks are acquired for existing nodes up to level n. If necessary, all nodes below level n are locked by a subtree lock (SR, SX) at level n.

Let us highlight by three scenarios taDOM's flexibility and tailor-made adaptations to XML documents as compared to competitor approaches. Assume transaction *T1* – after having set appropriate intention locks on the path from the root – wants to read-lock context node *cn*. Independently of whether or not *T1* needs subtree access, MGL only offers a subtree lock on *cn*, which forces concurrent writers (*T2* and *T3* in Fig. 9(a)) to wait for lock release in a lock request queue (LRQ). In the same situation, node locks (NR and NX) would allow greatly enhance permeability in *cn*'s subtree (Fig. 9(b)). As the only lock granule, however, node locks would result in excessive lock management cost and catastrophic performance behavior, especially for subtree deletion [20]. A frequent XML read scenario is scanning of *cn* and all its children, which taDOM enables by a single lock with special mode (LR). As sketched in Fig. 9(c), LR supports write access to deeper levels in the tree. The combined use of node, level, and subtree locks gives taDOM its unique capability to tailor and minimize lock granules. Above these granule choices, additional flexibility comes from lock-depth variations on demand – a powerful option only provided by taDOM.

8.2 The taDOM Protocol Family

Continuous improvement of these basic concepts led to a whole family of lock protocols, the taDOM family, and finally resulted in a highly optimized protocol called taDOM3+ (tailor-made for the operations of the DOM3 standard [8]),

which consists of 20 different lock modes and "squeezes transaction parallelism" on XML document trees to the extent possible. Correctness and, especially, serializability of the taDOM protocol family was shown in [21,40].

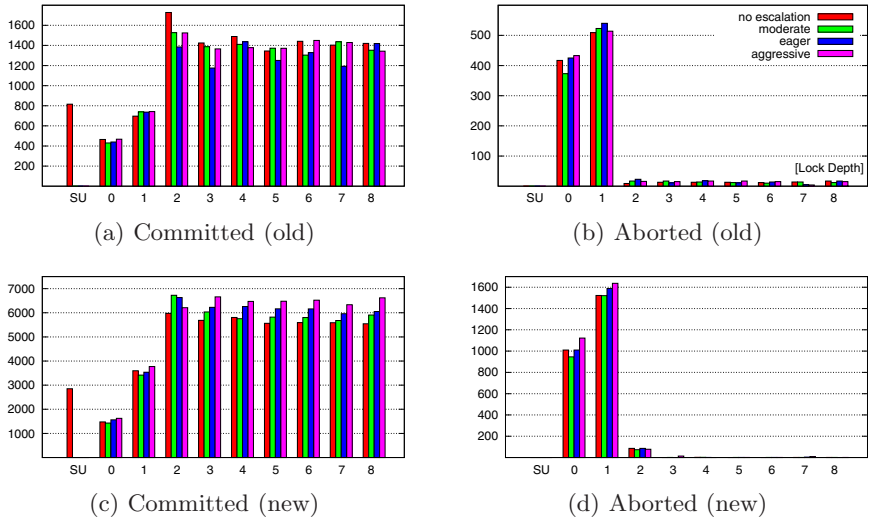
The concept of *meta-locking* implemented in XTC provides the flexibility to exchange lock protocols at runtime. Hence, such dynamic adaptations of lock management are a prerequisite to achieve workload-dependent optimization of concurrency control and to eventually reach autonomic tuning of multi-user transaction processing [2].

8.3 Enhancing Multi-user Performance

We cross-compared 12 protocols under identical workloads and in the same system environment [20] using meta-locking, i. e., without hardwiring all the different lock protocols in the XTC code. In this lock contest, the taDOM protocols have clearly proven their superiority over all competitors. Protocols only offering *node locks* were beaten roughly by a factor of 2 by MGL protocols which, in addition, provided *subtree locks*. Supplementary to MGL equipped with *level locks*, the taDOM protocol family, in turn, achieved once again a doubling of the transaction throughput [20].

Every improvement of the lock protocol, however, shifts the locking performance a bit more from the level of logical XML trees down to the underlying storage structures. Hence, an efficient and scalable B*-tree implementation in an *adjusted infrastructure* is mandatory. Together with fine-tuning measures to workload characteristics, we added the following drivers for locking performance to our initial XTC version:

- *B*-tree Locking (D1)*: Initial tree traversal locked all visited index pages to rely on stable ancestor paths in case of leaf page split or merges. Provoking high update contentions, we re-implemented our B*-tree to follow the ARIES protocol [33] for index structures, which is completely deadlock-free and can therefore use cheap latches (semaphores) instead of more expensive locks. Contention during tree traversals is reduced by *latch coupling*, where at most a parent page and one of its child pages are latched at the same time.
- *Storage Manager (D2)*: Needing full root-to-leaf traversal, navigation embodies a crucial performance aspect of a B*-tree-based storage manager. We observed high locality in the leaf pages and remembered those pages and their version numbers as a hint for future operations. Each time when reaccessing the B*-tree for navigation, we use this information to first locate the leaf page of the context node. Only if this hint fails, we have to perform a full root-to-leaf traversal of the index to find the correct leaf.
- *Buffer Manager (D3)*: Prefix-compression of SPLIDs is very effective to save storage space and disk IO, but must be paid with higher costs for encoding and decoding of compressed records. To avoid this unnecessary decoding overhead and to speed up record search in a page, we enabled buffer pages to carry a cache for already decoded entries.
- *Dynamic Lock Depth Adjustment(D4)*: Growing lock depth refines lock granules to minimal sizes that do not always pay off, because conflicting oper-



(a) Committed (old) (b) Aborted (old)

(c) Committed (new) (d) Aborted (new)

Fig. 10. Effects of lock depth and lock escalation on transaction throughput (tpm)

ations often occur at levels closer to the document root. In turn, it enlarges administration overhead, because the number of locks to be managed increases. Therefore, optimal lock depth depends on document properties, workloads, and other runtime parameters like multiprogramming level, etc., and has to be steadily controlled and adjusted at runtime. For this reason, we leveraged dynamic *lock escalation/deescalation* as the most effective solution. Using empirically proven heuristics for conflict potential in subtrees, the simple formula $threshold = k * 2^{-level}$ delivered escalation thresholds taking into account that typically fanout and conflicts decrease with deeper levels. Parameter k is adjusted to current workload needs.

– *Avoidance of Conversion Deadlocks (D5)*: Typically, deadlocks occurred when two transactions tried to concurrently append new fragments under a node already read-locked by both of them. Conversion to an exclusive lock involved both transactions in a deadlock. Update locks are designed for relational systems to avoid such conversion deadlocks [11], because they allow for a direct upgrade to an exclusive lock, when the transaction decides to modify the current record, or for a downgrade to a shared lock, when the cursor is moved to the next record without any changes. Transactions in XDBMSs do not follow such easy access patterns. Instead, they often perform arbitrary navigation steps, e. g., to check the content of child elements, before modifying a previously visited node. Hence, we carefully enriched our access plans with hints when to use update locks for node or subtree access.

Here, we can only sketch the results of these "performance drivers" which are described in [3]. We created read/write transaction benchmarks with high blocking potential, which access and modify a generated XMark document [37] at varying

levels and in different granules. To get insight in the behavior of the lock-depth optimization D4, we measured the throughput of transactions per minute (tpm) and ran the experiments for three escalation thresholds (moderate, eager, aggressive) in single user mode (SU) and in multi-user mode with various initial lock depths (0–8). To draw the complete picture and to reveal the dependencies to our other optimizations, we repeated the tests with two XTC versions: XTC based on the old B*-tree implementation and XTC using the new B*-tree implementation together with the optimizations D2 and D3. To identify the performance gain caused by D1–D3, we focused on transaction throughput, i. e., commit and abort rates, and kept all other system parameters unchanged. Fig. 10 compares the experiment results. In single-user mode, the new version improves throughput by a factor of 3.5, which again highlights the effects of D2 and D3. The absence of deadlocks and the improved concurrency of the latch-coupling protocol in the B*-tree (D1) becomes visible in the multi-user measurements, where throughput speed-up even reaches a factor of 4 (see Fig. 10(a) and (c)) and the abort rates almost disappear for lock depths > 2 (see Fig. 10(b) and (d)).

Deadlocks induced by the old B*-tree protocol were also responsible for the fuzzy results of the dynamic lock depth adjustment (D4). With a deadlock-free B*-tree, throughput directly correlates with lock overhead saved and proves the benefit of escalation heuristics (see Fig. 10(c) and (d)).

9 Conclusions

In this survey, we outlined performance-critical concepts and their implementation in XTC. By observing performance bottlenecks or inappropriate system behavior in early experiments, we could adjust numerous algorithms in XTC. But removing a bottleneck often revealed another one at a higher performance level. Hence, we had to iteratively and repeatedly improve XTC to reach the current system version mature in many aspects. As outlined, we have identified and are still identifying during this maturing process many performance drivers in various architectural layers. So far, we have often gained orders of magnitude in component speed-ups and, as a consequence, dramatic overall performance improvements. Future research will address further enhancements in autonomic system behavior [38] and energy efficiency by using flash disks and implementing energy-aware algorithms in specific XDBMS components.

References

1. Aguiar Moraes Filho, J., Härder, T.: EXsum – An XML Summarization Framework. In: Proc. IDEAS, pp. 139–148 (2008)
2. Bächle, S., Härder, T.: Tailor-Made Lock Protocols and Their DBMS Integration. In: Proc. EDBT 2008 Workshop on Software Engineering for Tailor-made Data Management, pp. 18–23 (2008)
3. Bächle, S., Härder, T.: The Real Performance Drivers Behind XML Lock Protocols. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA 2009. LNCS, vol. 5690, pp. 38–52. Springer, Heidelberg (2009)

4. Bruno, N., Koudas, N., Srivastava, D.: Holistic Twig Joins: Optimal XML Pattern Matching. In: Proc. SIGMOD, pp. 310–321 (2002)
5. Chen, T., Lu, J., Ling, T.W.: On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. In: Proc. SIGMOD, pp. 455–466 (2005)
6. Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D., Candan, K.S.: Twig^{2}Stack: Bottom-Up Processing of Generalized-Tree-Pattern Queries over XML Documents. In: Proc. VLDB, pp. 283–294 (2006)
7. Christophides, W., Plexousakis, D., Scholl, M., Tourtounis, S.: On Labeling Schemes for the Semantic Web. In: Proc.12th Int. WWW Conf., pp. 544–555 (2003)
8. Document Object Model (DOM) Level 2 / Level 3 Core Specification. W3C Recommendation (2004), `http://www.w3.org/TR/DOM-Level-3-Core`
9. Fontoura, M., Josifovski, V., Shekita, E.J., Yang, B.: Optimizing Cursor Movement in Holistic Twig Joins. In: Proc. CIKM, pp. 784–791 (2005)
10. Graefe, G.: Hierarchical Locking in B-Tree Indexes. In: Proc. German National Database Conf. (BTW). LNI, vol. 65, pp. 18–42. Springer, Heidelberg (2007)
11. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco (1993)
12. Haas, L., Freytag, J.-C., Lohman, G.M., Pirahesh, H.: Extensible Query Processing in Starburst. In: Proc. SIGMOD, pp. 377–388 (1989)
13. Härder, T.: XML Databases and Beyond – Plenty of Architectural Challenges Ahead. In: Eder, J., Haav, H.-M., Kalja, A., Penjam, J. (eds.) ADBIS 2005. LNCS, vol. 3631, pp. 1–16. Springer, Heidelberg (2005)
14. Härder, T., Haustein, M.P., Mathis, C., Wagner, M.: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data & Knowl. Eng. 60(1), 126–149 (2007)
15. Härder, T., Mathis, C., Schmidt, K.: Comparison of Complete and Elementless Native Storage of XML Documents. In: Proc. IDEAS, pp. 102–113 (2007)
16. Härder, T., Schmidt, K., Ou, Y., Bächle, S.: Towards Flash Disk Use in Databases – Keeping Performance While Saving Energy? In: Proc. German National Database Conf. (BTW). LNI, vol. 144, pp. 167–186. Springer, Heidelberg (2009)
17. Härder, T., Reuter, A.: Concepts for Implementing a Centralized Database Management System. In: Proc. Int. Computing Symposium on Application Systems Development, pp. 28–60. B.G. Teubner-Verlag (1983)
18. Haustein, M.P.: Fine-Granular Transaction Isolation in Native XML Database Management Systems (in German), Ph.D. Thesis, Univ. of Kaiserslautern, Verlag Dr. Hut, München (2006)
19. Haustein, M.P., Härder, T., Mathis, C., Wagner, M.: DeweyIDs – The Key to Fine-Grained Management of XML Documents. In: Proc. SBBD, pp. 85–99 (2005)
20. Haustein, M.P., Härder, T., Luttenberger, K.: Contest of XML Lock Protocols. In: Proc. VLDB, pp. 1069–1080 (2006)
21. Haustein, M.P., Härder, T.: Optimizing Lock Protocols for Native XML Processing. Data & Knowl. Eng. 65(1), 147–173 (2008)
22. Jagadish, H.V., Al-Khalifa, S., Chapman, A.: TIMBER: A Native XML Database. The VLDB Journal 11(4), 274–291 (2002)
23. Jiang, H., Wang, W., Lu, H., Yu, J.X.: Holistic Twig Joins on Indexed XML Documents. In: Proc. VLDB, pp. 273–284 (2003)
24. Jiao, E., Ling, T.W., Chan, C.Y.: *PathStack*¬: A Holistic Path Join Algorithm for Path Query with Not-Predicates on XML Data. In: Zhou, L.-z., Ooi, B.-C., Meng, X. (eds.) DASFAA 2005. LNCS, vol. 3453, pp. 113–124. Springer, Heidelberg (2005)
25. Li, C., Ling, T.W., Hu, M.: Efficient Updates in Dynamic XML Data: from Binary String to Quaternary String. VLDB J. 17(3), 573–601 (2008)

26. Loeser, H., Nicola, M., Fitzgerald, J.: Index Challenges in Native XML Database Systems. In: Proc. German National Database Conf. (BTW). LNI, vol. 144, pp. 508–523. Gesellschaft für Informatik (2009)
27. Lu, J., Chen, T., Ling, T.W.: Efficient Processing of XML Twig Patterns with Parent-Child Edges: a Look-Ahead Approach. In: Proc. CIKM, pp. 533–542 (2004)
28. Lu, J., Chen, T., Ling, T.W.: *TJFast*: Effective Processing of XML Twig Pattern Matching. In: Proc. WWW, pp. 1118–1119 (2005)
29. Mathis, C.: Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems, Ph.D. Thesis, Univ. of Kaiserslautern, Verlag Dr. Hut, München (2009)
30. Mathis, C., Härder, T., Haustein, M.P.: Locking-Aware Structural Join Operators for XML Query Processing. In: Proc. SIGMOD, pp. 467–478 (2006)
31. Mathis, C., Härder, T., Schmidt, K., Bächle, S.: XML Indexing and Storage: Fulfilling the Wish List (submitted, 2009)
32. Miklau, G.: XML Data Repository, `http://www.cs.washington.edu/research/xmldatasets`
33. Mohan, C.: *ARIES/KVL*: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In: Proc. VLDB, pp. 392–405 (1990)
34. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: *OrdPaths*: Insert-Friendly XML Node Labels. In: Proc. SIGMOD, pp. 903–908 (2004)
35. Özcan, F., Seemann, N., Wang, L.: XQuery Rewrite Optimization in IBM DB2 pureXML. Data Engineering Bulletin 31(4), 25–32 (2008)
36. Qin, L., Yu, J.X., Ding, B.: *TwigList*: Make Twig Pattern Matching Fast. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 850–862. Springer, Heidelberg (2007)
37. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In: Proc. VLDB, pp. 974–985 (2002)
38. Schmidt, K., Härder, T.: Usage-Driven Storage Structures for Native XML Databases. In: Proc. IDEAS, pp. 169–178 (2008)
39. Schöning, H.: *Tamino*–A DBMS Designed for XML. In: Proc. ICDE, pp. 149–154 (2001)
40. Siirtola, A., Valenta, M.: Verifying Parameterized taDOM+ Lock Managers. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 460–472. Springer, Heidelberg (2008)
41. Weiner, A.M., Härder, T.: Using Structural Joins and Holistic Twig Joins for Native XML Query Optimization. In: Grundspenkis, J., Morzy, T., Vossen, G. (eds.) ADBIS 2009. LNCS, vol. 5739, pp. 149–163. Springer, Heidelberg (2009)
42. Weiner, A.M., Härder, T.: A Framework for Cost-Based Query Optimization in Native XML Database Management Systems. In: Li, C., Ling, T.W. (eds.) Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization, and Data Query Technologies. IGI Global (2010)
43. XML on Wall Street, Financial XML Projects, `http://lighthouse-partners.com/xml`
44. Yu, J.X., Luo, D., Meng, X., Lu, H.: Dynamically Updating XML Data: Numbering Scheme Revisited. World Wide Web 8(1), 5–26 (2005)
45. Yu, T., Ling, T.W., Lu, J.: *TwigStackList¬*: A Holistic Twig Join Algorithm for Twig Query with Not-Predicates on XML Data. In: Li Lee, M., Tan, K.-L., Wuwongse, V. (eds.) DASFAA 2006. LNCS, vol. 3882, pp. 249–263. Springer, Heidelberg (2006)