Locking-Aware Structural Join Operators for XML Query Processing

Christian Mathis, Theo Härder, Michael Haustein
Dept. of Computer Science, University of Kaiserslautern, Germany
{mathis | haerder | haustein}@informatik.uni-kl.de

ABSTRACT

As observed in many publications so far, the matching of twig pattern queries is a core operation in XML database management systems (XDBMSs) for which the structural join and the holistic twig join algorithms were proposed. In a single-user environment, especially the latter algorithm provides a good evaluation strategy. However, when it comes to multi-user access to a single XML document, it may lead to extensive blocking situations: The XDBMS has to ensure data consistency and, therefore, has to prevent concurrent modification operations from changing elements in the input sequences, a holistic twig algorithm accesses while operating. To circumvent this problem, we propose a set of new *locking-aware* operators for twig pattern query evaluation that rely on stable path labeling IDs (SPLIDs) as well as document and element set indexes. Furthermore, by running empirical tests on a native XDBMS, we show that their performance is comparable to existing approaches in a single-user environment, and leads to higher throughput rates in the case of multi-user access.

1. MOTIVATION

As XML documents permeate information systems and databases with increasing pace, they are more and more used in a collaborative way. The challenge for database system development is to provide adequate and fine-grained management for these documents enabling efficient and concurrent read and write operations. In essence, this objective postulates the design and management of highly dynamic XML documents. Because event-driven, navigational, and declarative languages are already available in the form of (partial de-facto) standards like SAX, DOM, XPath, or XQuery [21], and used as typical XML document processing (XDP) interfaces, XDBMSs should be able to run concurrent transactions supporting all these interfaces simultaneously and, at the same time, guarantee ACID properties [10] for all of them.

Multi-lingual XDP support explicitly means that—starting from a context node—navigational operations of the DOM language model such as *parent/first-child/last-child/previous-sibling/next-sibling* and event triggering for SAX nodes in *document order* must be facilitated and, at the same time, adequate support for declarative queries of the XQuery and XPath 2.0 language model must be guaranteed. Furthermore, transaction-safe modifications in any part of the document must be enabled.

ACM SIGMOD 2006, June 26-29, 2006, Chicago, Illinois, USA

In a first step, we concentrate on twig queries, i.e., queries that only contain the child and descendant axes and allow branching path-based predicates (Figure 4b). They are often exploited in XML query processing by decomposing them into sequences of operations where tailored axes operators are employed and chained together to derive the final result. Another evaluation approach is the application of holistic twig join algorithms (twig joins for short).

1.1 Required System Support

Currently, a growing wave of proposals for index structures is published for XDBMS use. They can be classified into structure, content, and hybrid indexes [4] where the latter two classes borrow many ideas from well-known Information Retrieval approaches. Because they focus on node access based on values, they are of limited use for structural processing steps required for our query patterns sketched above. In contrast, structural access support is of prime importance. Therefore, we claim that such a mechanism should facilitate all path processing steps in an orthogonal way, thereby providing dynamic reorganization, balanced structure, and logarithmic access costs.

Together with appropriate index structures, a labeling scheme for tree nodes is most influential for efficient access to XML documents. It should support all the navigational operations as well as the evaluation of the main axes for declarative query processing. At the same time, the labeling scheme must facilitate the work of the lock manager which needs stable identifiers—similar to TIDs in the relational world—to lock individual nodes and subtrees of a document. Note, fine-grained locking of an arbitrary tree node also requires locks in appropriate modes on all its ancestor nodes to be kept until transaction commit. Therefore, dynamic relabeling of nodes (as a consequence of a subtree insertion) would not only be very expensive because (a part of) the document residing on disk has to be modified, but would also strongly interfere with lock management. For these reasons, node labels should *be immutable* (for the life time of the nodes), should easily reveal *the level* and *the IDs of all ancestor nodes*, and must, when inserting new nodes, *preserve the document order*. When storing an XML document, the *round-trip property* must be guaranteed, that is, the XDBMS must be able to reconstruct the *original* ordered document transferred by a client application. Last, but not least, the labels need a very efficient variable-length representation, because there are frequently millions of nodes to process in large XML documents.

1.2 Our Contribution

We believe that only selected indexing and labeling schemes as well as query processing approaches can fulfill the strong requirements outlined above. The interplay of these concepts proposed so far was not considered in detail; in particular, the needs of locking protocols for document modifications together with related index maintenance was not explored. Furthermore, none has taken the support of navigation into account which can be optimized together with the physical document mapping.

Existing query evaluation algorithms (for so-called *twig queries*) are running sequences of processing steps where single-user environments are assumed. They have to lock large element sets to avoid, for example, phantoms (see Section 2.3)—thereby blocking them against modifications in multi-user mode—, even if the result set is small in case of highly selective queries. Hence, they are not aware of the large data granules to be locked until transaction commit to guarantee transaction isolation levels *repeatable read* or even *serializable* [10]. Therefore, these algorithms cannot respond to and optimize complex transaction mixes typically occurring in multi-user environments. For this reason, we develop alternative—so-called *locking-aware*—path processing operations, in particular, structural join algorithms which are sensitive to the requirements of multi-lingual XDP interfaces and concurrent transactions. As a result, we propose a set of physical algorithms supporting processing steps of XML query evaluation as kind of locking-aware building blocks to be selected by the query optimizer.

We claim that the key to fine-grained and efficient management of XML documents are special—for dynamic documents optimized—prefix-based labeling schemes (as i.e. explored in [11]). Our algorithms take advantage of their expressiveness and our experimental results provide indicative hints of their potential.

In this paper, Section 2 gives an overview of the essential internals of our native XDBMS prototype XTC [12] implemented in Java; some numbers drawn from our extensive empirical experiments are characterizing the power of the labeling schemes mentioned above. Section 3 discusses the locking support needed for twig queries. Our approach to locking-aware path processing based on these system internals is outlined in Section 4; here we explore methods to efficiently implement twig joins. In Section 5, we describe some empirical experiments giving evidence of the performance potential of our approach. Finally, we wrap up with conclusions.

2. SYSTEM TESTBED

Our XTC system adheres to the well-known layered hierarchical architecture: The concepts of the storage system and buffer management could be adopted from existing relational DBMSs. The access system, however, required new concepts for document storage, indexing, and modification including locking. The data system available only in a slim version is of minor importance for our considerations.

2.1 Path Labels

A comparison and evaluation of node labeling schemes in [11] recommends node labeling schemes based on the Dewey Decimal Classification [6]. The abstract properties of Dewey order encoding—each label represents the path from the document's root to the node and the local order w.r.t. the parent node; in addition, sparse numbering facilitates node insertions and deletions—are described in [20]. Refining this idea, a number of similar labeling schemes were proposed differing in some aspects such as overflow technique for dynamically inserted nodes, attribute node labeling, or encoding mechanism. Examples of these schemes are ORDPATH [18] (used in MS SQL Server), DeweyID [15], or DLN [3]. Because all of them are adequate and equivalent for our processing tasks, we prefer to use the substitutional name *stable path labeling identifiers* (SPLIDs) for them.

General properties are the following: Each node label contains the label of its parent node as a prefix. A node label consists of a sequence of so-called *divisions* (separated by dots in the human-readable format). Odd division values indicate a level transition whereas even division values provide an overflow mechanism. Upon initial document storage, only odd division values are assigned, e.g., d_1=1.3.3 and d_2=1.3.5 label two consecutive nodes at level 3. A later insertion of a node at level 3 before d_2 receives the label d_3=1.3.4.3, which allows for correct level identification by counting simply the number of odd values, order preservation, node label comparison (e.g., $d_3<d_2$), and ancestor determination (e.g., 1.3 and 1) without relabeling the nodes. Division value 1 at levels > 1 is used to label attribute nodes (where order does not matter). An effective way to handle later insertions and, at the same time, to avoid overflows is to provide for gaps in the labeling space, that is, to initially assign the division values *dist+1*, *2*dist+1*, etc. where the parameter *dist* governs the gap size. The minimum value *dist=2* should be applied to almost static XML documents whereas larger *dist* values avoid resorting too frequently to overflow values; however, large *dist* values increase the storage space needed for the SPLIDs encoding.

Here we can only summarize the benefits of the SPLID concept; for details, see [3, 11, 18]. It provides holistic system support. Existing SPLIDs are immutable, that is, they allow the assignment of new IDs without the need to reorganize the IDs of nodes present. In theory, SPLIDs are free of maintenance under arbitrary insertions. Yet, implementation restrictions (e.g., key length < 128B in B-trees) may enforce subtree relabeling, either by exclusive reorganization or dynamically by a concurrent transaction (potentially aborting the violating one before). But worst-case experiments indicate that such events are very infrequent [14]. All SPLID properties are preserved and (equally important) relabeling only concerns the subtree.

Comparison of two SPLIDs allows ordering of the related nodes in document order. As opposed to competing schemes, SPLIDs easily provide the IDs of all ancestors to enable intention locking of all nodes in the path up to the document root without any access to the document itself. Declarative queries are supported by the efficient evaluation—that is, computation without the need to access the document on disk—of the eight axes frequently occurring in XPath or XQuery path expressions: *parent/child, ancestor/descendant, following-sibling/preceding-sibling, following/preceding.* Even sequential document processing and navigational operations to parent/child/sibling nodes from the context node are facilitated in combination with the storage structures sketched in the following.

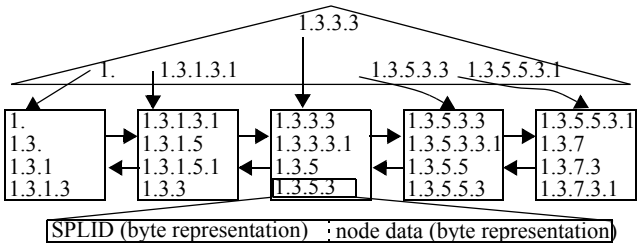


Figure 1. Document store with a B-tree and container pages

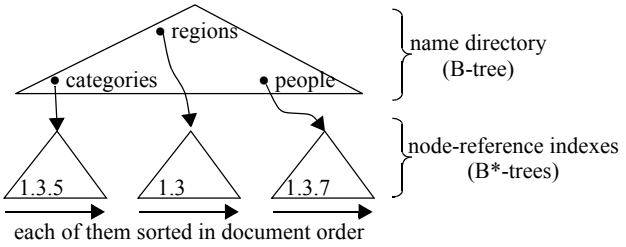


Figure 2. Organization of the element indexes

2.2 Fine-Grained Access to XML Documents

Our solution includes indexed access, maintenance of document order, variable-length node management, and representation of long fields. As sketched in Figure 1 and Figure 2, B-trees and B*-trees embody an efficient storage framework for XML trees and provide indexed access and order maintenance for them. Variations of the entry layout for the nodes allow for single-document and multi-document stores, key compression, use of vocabularies, and specialized handling of short documents [12].

Figure 1 illustrates the storage representation of a document; a B-tree, the so-called *document index*, with key/pointer pairs (SPLID+PagePtr) references the first node in each page of the *document container* consisting of a set of doubly-chained pages.[1] These pages contain the node data where text values exceeding a given threshold are stored in referenced mode. The document order obviously facilitates *first/next child* navigation. Locating the *parent* or *next/previous sibling* by a sequential scan in the container pages could possibly provoke a bad performance, because a (potentially) large set of pages had to be traversed; by taking the SPLID of the current node as a "hint", the document index always limits this operations to a single container page access. For example, given 1.3.7 as the current node, the previous sibling 1.3.5 is efficiently located via the document index, no matter how many container pages are in between. Using 16K pages, the document index is usually of height 1 or 2. Because of reference locality in the B-tree while processing XML documents, most of the referenced tree pages are expected to reside in DB buffers—thus reducing external accesses to a minimum. Having the magnitude of 10^8 nodes in 10^5 pages, document containers need careful optimization considerations. All node formats (of type element, attribute, or text) are of variable length. An element node only consists of a key and a name part, whereas an attribute node has a value part in addition. In contrast, a text node has only a key and a value part. Because the key part—consisting of a one-byte field KL (key length) and the encoded SPLID—may become the Achilles heel of the storage representation, it must be reduced in a very efficient way.

In addition to the document store, an *element index* is created consisting of a *name directory* with (potentially) all element names occurring in the XML document (Figure 2); this name directory often fits into a single page. For each specific element name, in turn, a *node-reference index* may be maintained which addresses the corresponding elements using their SPLIDs. Conceptually, these indexes are sequential lists which may contain very large numbers of SPLIDs; organized as B*-trees, they enable direct access. In all cases, variable-length key support is mandatory; prefix compression of SPLID keys turns out to be very effective.

[1] If we prefix the SPLIDs with the identifier of the individual documents, we obtain a multi-document store where the documents can be easily partitioned into separate sets of container pages.

2.3 Locking Documents and Indexes

Processing XML trees in multi-user environments requires sophisticated isolation mechanisms to perform scanning, navigation, declarative read operations and arbitrary element/subtree insertions/deletions. Often, XDBMS interface operations from DOM implementations are processed directly on the document nodes where several concurrently running transactions have to be isolated. At the same time, declarative queries—written in XPath or XQuery—are using indexed node references for the evaluation of path processing steps and, potentially, additional navigation steps thereby traversing the document to complete their query results.

Simply transferring the proven isolation techniques known from (object-)relational systems to XDBMSs leads to several drawbacks in concurrent transaction handling.[2] For example, evaluations of the child axis or navigation steps from node to node are frequently applied during XML query processing, but multi-granularity locking [10] only provides for intention and subtree locks. Hence, it is unable to lock an entire document level or a single XML node without affecting the nodes residing in the related subtrees. Protocols using latches to synchronize read and write operations on tree structures maximize the processing throughput by sophisticated algorithms, but they just lock the required tree sections and unlock the no longer needed sections as soon as possible. As a consequence, these latches only isolate single operations and do not take any transactional scope into account.

To cope with the requirements for transaction isolation, we implemented a hierarchical lock protocol [12, 13] which provides tailored lock modes for tree operations and tuning options such as lock depth. It synchronizes the fine-grained shared and exclusive access to single XML nodes, their potential attributes and values, and access to entire document levels or subtrees. When a pre-specified lock limit is reached, lock escalation may be applied, such that no running transaction has to be aborted for reasons of resource unavailability. Query evaluation is typically performed by locating

[2] Using timestamped snapshots for reading and copies of subtrees for updates is e.g. applied in SystemRX [2]. For high transaction throughput, this requires the maintenance of several copies affecting clustered XML store and enforces blocking of even larger parts to enable document re-clustering (a premise for high-performance DBMSs).

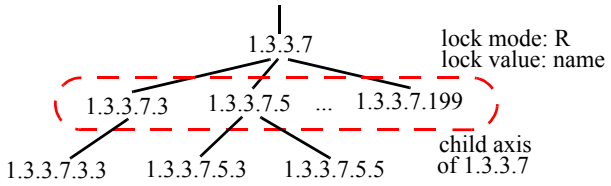


Figure 3. Sample lock situation for the child axis

XML elements via indexes; this means that lock acquisition for arbitrary context nodes and all their ancestor nodes up to the document root is a very frequent operation. To provide effective support, in turn, the calculation of all these node IDs must be performed in memory without accessing the nodes in the document stored on disk. In particular, as a result of addressing the nodes by SPLIDs, our lock protocols can operate very efficiently.

Furthermore, read and write operations affecting the update of index structures have to be synchronized, too. Therefore, we combine the node-based hierarchical locking with a kind of key-range index locking [17] extended by XPath axes semantics (called *axis locks* in this paper). Axis locks requested for document modifications and element index accesses prevent the occurrence of any phantom in the specified axis, while navigational and declarative queries are processed concurrently. As a brief example, an axis lock can be requested in read mode for the context node with SPLID 1.3.3.7, the lock value *name* and its axis *child*. This lock prevents any concurrent transaction from inserting or deleting a node with value *name* that resides in the *child* axis of the node 1.3.3.7. But the lock does not restrict parallel transactions more than necessary, because *name* nodes can be inserted into any other axes areas of the document (e.g., as descendants of 1.3.3.7 except children or in disjoint subtrees).

An example situation for the described child axis lock on the node with SPLID 1.3.3.7 is shown in Figure 3. Because of the locked value *name* in mode *R* the node 1.3.3.7.5 can be updated if its value differs from *name*, otherwise such an operation would be blocked. The same applies for the insertion of 1.3.3.7.199 which can only be performed if the node value is not *name*. Deeper levels (e.g., the level containing 1.3.3.7.3.3 or 1.3.3.7.5.3) are not affected by this lock and the existing nodes are allowed to be updated or deleted or new nodes can be inserted.

2.4 Empirical Results of SPLID Use

Implementation of SPLIDs has to be done carefully because of their potential size primarily influenced by the document depth, the node fan-out, and the *dist* parameter. Therefore, serious efforts are needed to optimize their representation and storage to provide for a practical solution.

In [11], we report on the results of a comprehensive empirical study of SPLID use for node labeling of XML trees. Under a wide variation of sizes and shapes, we obtained quite stable results for the storage consumption of SPLIDs. We used a Huffman encoding scheme primarily because of its ability to optimize the codes to the frequency distributions of the division values occurring in the set of node labels for a document. By representing a SPLID as a sequence of divisions, we have systematically varied the *dist* parameter to cover the reasonable solution space for practical use. For example, using *dist=2* and *dist=32* as parameter values, average SPLID sizes ranged from ~3 to ~8 resp. ~5 to ~12 bytes.

a) **Queries**

Q1)//book[title="Momo"]//author["Ende"]

Q2) **for** $b **in** //book, $a in $b//author
where $b/title="Momo" and $a="Ende"
return ($b, $a)

b) **Twig**

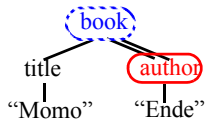


Figure 4. Sample queries and a twig

As visualized in Figure 1, the document order inherent in the sequence of SPLIDs lends itself to prefix compression. Therefore, we obtained the favorable result that the average size of prefix-compressed SPLIDs for all documents under a wide variation of the *dist* parameter considered varied between ~2 and <3 bytes. Applying prefix compression to the element indexes, the average compressed SPLID—although indexed in document order—did not reach the reduction rate of the document store. Nevertheless, we again obtained acceptable prefix-compressed results from ~3 bytes to ~6 bytes. As a rule of thumb, we may expect in all cases average SPLID sizes which are comparable to TIDs and can therefore be processed in similar ways both in memory and on disk.

3. TWIG QUERY MATCHING

Because XML data has a tree-structured data model, it is very natural to use path and tree patterns for the search of structurally related XML elements. Therefore, expressions specifying those patterns are a common and frequently used idiom in many XML query languages and their fast and effective evaluation is of utmost importance for every XML query processor. A particular pattern instance—the *twig*—has gained much attention in recent publications, because it represents a small but practical class of queries, for which effective evaluation algorithms have been found [4].

3.1 Twig Queries

Basically, a twig is a tree $QT = (V, E, \lambda, r)$ where V is a set of nodes, $E \subseteq V \times V$ is a set of edges, λ is a mapping $\lambda : E \rightarrow \{child, descendant\}$, and r is the root node of the tree. Figure 4b depicts such a twig. The nodes represent simple predicates, such as tests on element names or content, while edges express the desired relationships between the elements to be found (in the graphical notation, we use the double line for the descendant relationship and the singleton line for the child relationship). Thus, the twig represents a class of queries that only contains child and descendant operators.

The specific problem of twig query evaluation is to find all possible embeddings for a given twig in an XML document such that, 1) for each embedding, a node of the twig corresponds to exactly one XML element with the same name and vice versa, and 2) the structural relationships between the XML elements found exactly fulfill those defined between the nodes of the twig. The result of a twig

evaluation may be represented by a sequence of tuples. For example, the twig from Figure 4b, evaluated on the XML document whose fragment is shown in Figure 5, returns a sequence of tuples with the name fields [*book, title, "Momo", author, "Ende"*] and the values [1.3, 1.3.3, 1.3.3.3, 1.3.5.3, 1.3.5.3.3], [1.3, 1.3.3, 1.3.3.3, 1.3.7.3, 1.3.7.3.3], and so on. Please note that these two tuples depicted only differ in the elements *author* and "*Ende"*. Because the element *book* (1.3) has two such elements as descendants, all remaining elements matched by the twig have to be unnested and, therefore, repeated.

3.2 Twig Algorithms and Locking Support

A large class of effective methods for twig query evaluation builds on two basic ideas: the *structural join* (or containment join) [1, 22] and the *holistic twig join* [4]. The first approach decomposes the twig into a set of binary join operations, each applied to neighbor nodes of the twig. For example, the relationship between the nodes *item* and *name* may be evaluated by 1) accessing all elements with the name *item* through a cursor C_1 (e.g., by a scan over the leaf pages of the corresponding node reference index, see Section 2.2), 2) accessing all elements with the name *name* through C_2, and 3) joining these two sequences by a single iteration over C_1 and C_2 using the parent-child relationship as the join predicate (which can easily be done with SPLIDs). Intermediate results derived by the various structural joins in the tree are then—to use the term of the authors in [1]—"stitched" together to produce the final result. The underlying algorithm implementing the structural join operator is interchangeable and subject to current research. [22] proposes a merge join (the so-called Multi-Predicate Merge Join, MPMGJN), while [1] introduces the Stack-Tree algorithm.

In [4], the authors argue that, intrinsic for the approach above, intermediate result sizes may get very large, even if the final result is small, because the intermediate result has to be unnested for the structural join approach, too, as outlined in the twig example above. As a consequence, in the worst case, the size of an intermediate result sequence is in the order of the product of the sizes of the input sequences. In addition to this problem, there is the need for a cost-based query optimization to find the optimal join order. To remedy these obstacles, the twig join (TwigStack) [4] evaluates the twig as a whole: For each twig node *n* Stack S_n is initialized, as well as a cursor C_n, which provides access to all elements that fulfil the given node predicate p_n. As in the structural join approach, the algorithm iterates over the cursors to find twig matches, but avoids intermediate result unnesting by suitably encoding the qualified elements on the set of stacks.

Crucial to the performance of these algorithms is the way, elements are accessed through the cursors. Often, large ranges of elements may be skipped, because the state of other cursors allows the calculation of the next possible match in C_n. This idea is exploited by *TS-Generic+* using a special index structure [15] and by the concept of *virtual cursor movements* in the *TwigOptimal* algorithm [8]. For the correct evaluation of the query, these algorithms aim to only transfer as few as needed nodes from external memory.

When running such (skipping) twig join algorithms in multi-user context, where concurrent transactions may insert/update/delete arbitrary elements/subtrees in the queried XML document, appropriate locking support is needed. For isolation level *repeatable read*[3] [10], node locks (see Section 2.3) on all physically accessed elements suffice, because these elements form a superset of all result elements, whose stability has to be assured. Therefore, in this case, the locking characteristics for twig query evaluation (number of required locks; size of locked document ranges and, thus, the possibility for concurrent modifications) heavily benefits from the intention of the proposed twig join algorithms to skip as large ranges of elements as possible.

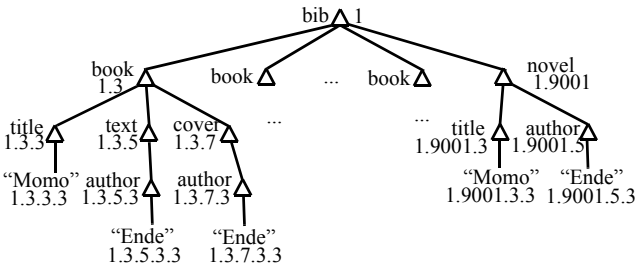


Figure 5. XML document (cut-out)

On the other hand, when prevention of phantoms is required in isolation level *serializable* [10], different measures have to be taken. Suppose, at first transaction T_1 evaluates Q1 from Figure 4 against a document, receives a result sequence R_a but does not commit. Then T_2 changes the name of a *novel* element (with title "Momo" and author "Ende") to *book* and commits, thus creating a new match for the query in T_1. Finally T_1 poses the same query again and receives R_b. Using only node locks for T_1 in this situation would not prevent T_2 from changing the novel element, because the new book element did not exists before and could therefore not be locked. The repeated evaluation of Q1 yields a different result sequence ($R_a \neq R_b$) containing a phantom.

3.2.1 Solution 1: Document-Wide Axis Locks

A solution for this situation is the use of axis locks (see Section 2.3): For T_1 an axis lock is granted on the root element of the document with the axis descendant-or-self for each node name in the twig query (*document-wide axis lock*) *before* the query is evaluated. Because T_1 issues (among others) an axis lock for all book elements in the document in our example, T_2 is not allowed to alter the set of book elements (neither by insertion nor by deletion). This solution is independent from the strategy (algorithm) used to evaluate the twig query. The correctness of this approach follows from Theorem 1.

Theorem 1: The result sequence of a twig query relying on the element sequences (cursors) $C_1...C_n$ in a transaction T_1 may only be affected by a writer transaction T_2, iff T_2 at least modifies one of the element sequences accessed and vice versa. (Proof obvious).

[3] Repeatable read guarantees that queries issued multiple times within a transaction always get the same elements, i.e., no element accessed may be modified concurrently. However, in contrast to isolation level *serializable* repeatable read does not avoid phantoms.

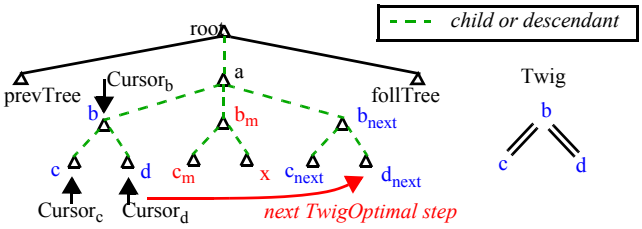


Figure 6. Snapshot of a twig being evaluated

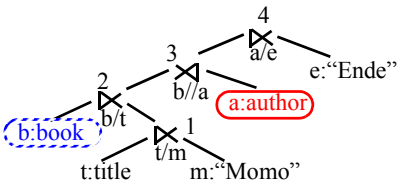


Figure 7. Operator plan for query Q1

3.2.2 Solution 2: LCA Locking

Another possibility is the generation of local (i.e., non-document-wide) locks, *while* the query is processed. This, however, depends on the evaluation algorithm *A* and its strategy to access elements. Suppose *A* is a skipping holistic twig join algorithm (like TwigOptimal) and consider Figure 6, where a snapshot of *A* processing a twig is depicted. In this situation, *A* could issue the following actions: 1) $Cursor_d$ is moved to d_{next}, and after an analysis of d_{next}'s relatives, 2) $Cursor_b$ and $Cursor_c$ would be moved to b_{next} and c_{next} (by skipping b_m and c_m). Regarding the lock protocol, we have to ask *how is a concurrent transaction prevented from renaming a node (e.g., x) to d—thus generating a phantom*. Let us assume that we correctly solved this problem for all nodes in document order ahead of the current cursor. To prevent element *x* from being concurrently modified, we have to make sure that range *R* between $Cursor_d$'s source *s* and target *t* is locked. This can only be accomplished by an axis lock, because we cannot compute any nodes in *R* to be protected by a node lock. The only meaningful and possible solution is to place a descendant-or-self axis lock on the least common ancestor (LCA) of *s* and *t*[4] (element *a* in the example). To complete this lock strategy, a rule for the initial cursor setup has to be defined: On each initial cursor node c_{1i}, a preceding and an ancestor-or-self lock has to be acquired to avoid phantoms on elements in document order before c_{1i}.

Obviously, both approaches constrain concurrent document modifications more than necessary. In fact, both strategies behave similarly: for LCA Locking, it takes only two nodes of the same name occurring on two distinct paths in the document to place a document-wide axis lock for that name on the root node; and this situation may occur frequently. However, when only taking node locks and axis locks into account (and no further meta-data, e.g., structural summaries), the so-far proposed solutions are the only possibilities to guarantee serializability for holistic twig join algorithms.

4. LOCKING-AWARE STRUCTURAL JOIN OPERATORS

The proposed methods for twig evaluation have led to a stepwise improvement in performance over the time, and they are now good candidates for physical operators supporting twig query evaluation in XDBMSs. However, for our quest to find a more flexibly applicable set of physical operators, especially considering the background of phantom prevention, let us rethink the process of twig evaluation. Several key requirements strongly influenced the development of our structural join operators in Section 4.3.

4.1 Key Requirements

Use pipelined query processing. Pipelining, based on the ONC protocol (open, next, close) is a well-known strategy providing simple data and process flow mechanisms as well as low main memory usage, and assuring fast production of first result tuples. Our protocol additionally dictates that each semi-join operator receives a sequence of elements in document order as input and provides its resulting elements also in document order and as early as possible.

Design single-pass algorithms. To provide linear worst-case runtime behavior, our algorithms should not read (even parts of) input sequences multiple times.

Exploit the fact that not all nodes in the twig contribute to the query result. As observed in [8], most practical queries contain only some twig nodes (*extraction points*) for which output has to be generated (e.g., the *author* node for the XPath query and additionally the *book* node for the XQuery expression). With knowledge about extraction points, the query optimizer is able to pick semi-join algorithms (in many cases) instead of full joins for the generation of an operator plan. For example, consider the operator plan in Figure 7 which embodies one way to evaluate the twig in Figure 4. After we join the *title* elements with the content elements "Momo", the latter ones are not needed anymore for the evaluation of the rest of the query. Therefore a semi-join suffices. Note, if we wanted to evaluate the twig for the XQuery expression, then the join operator between *book* and *author* had to be a full join, because otherwise, we would discard the *book* information at this stage of evaluation and could not generate any output for books in the end.

The use of semi-joins has several benefits. Obviously, the intermediate result size is reduced in two ways: Because only relevant tuples are forwarded to the next operator, the absolute (byte) size is reduced. According to the XPath and XQuery standards, the result sequences of our operators are supposed (and can be enforced by our algorithms) to be tuplewise duplicate free. Therefore, the worst-case complexity of the result size is linear to the size N of the forwarded input list (as opposed to full join operators whose complexity is $O(N_1*N_2)$). Due to restricted space, we will only focus on algorithms for semi-joins in the remainder of this paper, although we found similar techniques as proposed in Section 4.3 for full joins.

Avoid duplicates. Some XPath axes like parent, ancestor, and descendant produce duplicates depending on the document structure

[4] This may be verified by considering all combinations of axis locks (self, parent, child, anc(-or-self), desc(-or-self), prev-sibl., foll-sibl.) and all computable SPLIDs on which the lock may be set (ancestors of all so far accessed elements). Note, the axes previous and following are prohibited, because they lock the complete document, which is not considered "meaningful".

and the input sequence(s). Reference [9] observes that duplicates occurring in intermediate results of an XPath query evaluation may generate an exponential worst-case behavior depending on the size of the query. One possible solution is the application of duplicate removal operators after each "critical" step. However, because duplicate removal is an expensive operation, we want our new operators to generate a duplicate-free output for duplicate-free input.

Design symmetric algorithms. This consideration is related to the internal element access strategy of each operator (see Section 4.3). Join operator 1 in Figure 7, for example, could be evaluated by 1) accessing all title elements and 2) checking for each *title* element *t* whether or not *t* contains text element "Momo". Algorithms using this method simply *filter* their input sequence. Therefore, they are called "Filters". An alternative evaluation strategy accesses all "Momo" text elements first and similarly performs an element-at-a-time lookup using an index to generate the sequence of *title* elements which actually are parents of a text element "Momo". Because—from the viewpoint of the initially scanned (text) element sequence—the operator performs a step towards the parent sequence, we call this class "Step" operators. The decision which operator to choose for plan generation has to be made by the query optimizer and depends on the statistical data distribution.

The requirement for symmetric algorithms is also crucial to enable *join reordering*. When using only full join operators, this issue is not a problem, because a result tuple always contains both, the parent/ancestor and the child/descendant information of a structural relation. For semi-joins, however, our operators have to explicitly calculate the reverse axes ancestor/parent of the axes descendant/child occurring in a twig query to support join reordering. As a convention, we will use the terms *upward* and *downward step* for the parent/ancestor and the child/descendant evaluation modes, respectively, and *top filter* or *bottom filter* depending on the input sequence filtered. Furthermore, the letter A will refer to the sequence of possible ancestors/parents, whereas we use B for children/descendants.

Avoid locking of complete element sequences. To relieve the blocking situations caused by the lock protocols needed, our new algorithms should access as small data granules as possible and therefore should be aware of the lock granules implied. For this purpose, we exploit SPLIDs and the document and element indexing techniques sketched in Section 2.2. The basic idea is to initially scan, just as TwigStack does, an (ideally small) element sequence via the element reference index, but only for one twig node. For this access, a document-wide axis lock is needed to prevent the scanned element sequence from concurrent modification. However, because this sequence is supposed to be small, e.g., the element sequence for a very selective predicate, concurrent modifications are only slightly hindered. The other twig nodes are then processed via element-at-a-time lookup in the document or element index, thus acquiring only node locks for *upward steps* and *bottom filters* and local axis locks for *downward steps* and *top filters*. The resulting lock distribution is then sufficient to guarantee serializability.

For example, query Q1 in Figure 4 may be evaluated in transaction T_1 using the operator plan from Figure 7 on the document in Figure 5 in the following steps: a) All text elements with the value

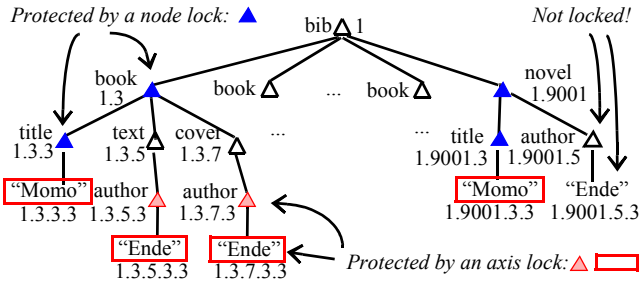


Figure 8. Lock state after evaluation of Q1

"Momo" are scanned via an index access and locked with a document-wide axis lock[5]. Now, a transaction T_2 is not able to insert the value "Momo" into the document. However, the novel element may still be changed to book. This is no problem, because, if T_2 commits before T_1 reaches any book element, T_1 will find a regular match. Otherwise, if T_2 is still active, T_1 has to wait for its commit because of the exclusive node lock on the new book. b) For each scanned text element, the SPLID of the parent element is calculated and accessed via the document index (upward step), causing a node lock for this element. Of course, all elements having other names than *title* are filtered out. In this state, *title* elements may still be inserted, but it is not possible, to rename an existing parent of a "Momo" text element to title, because it has a node lock. c) Using the same technique (upward step) as in b), we navigate to the *book* elements. Now T_2 is not able to rename the *novel* element anymore (but other elements, which are unrelated to the query of T_1 may still be modified). d) For each *book* element found, we query the element reference index for all descendant elements with the name *author*, thus generating a local axis lock for all intermediate *book* elements and the descendant axis (downward step). e) In the last step, we use the same technique to filter out those elements that have not value "Ende" as author (top filter). In this state (see Figure 8), a concurrent transaction cannot generate a phantom. All elements reached by an upward step, such as *titles* and *books,* are protected by a node lock, whereas all elements reached via the node reference index in a downward step (or top filter) are protected from modification by local axis locks. If a conflict situation would occur while T_1 processes the query, either T_1 or the other conflicting transaction would have correctly been blocked as the following theorem states:

Theorem 2: The strategy to lock the initial and all downward directed joins (top filter/downward step) by axis locks and all upward directed joins (upward step/bottom filter) by node locks guarantees isolation level serializable for the evaluation of a twig query Q.

Proof (Idea): Theorem 2 can be proven by induction over the sequence of joins needed. Each intermediate processing state is the result of a part (subtree) of *Q*, for which we assume that it is correctly locked. For the next join S, we can show that a concurrent transaction may not generate any phantom before, while, and after S is in process. ■

[5] In the current development phase of our XTC system, no content indexes are available. However, the extension of the proposed concepts to support content queries is straightforward.

4.2 A Classification of Semi-Join Algorithms

By examining the operator plan in Figure 7 we can infer three (mainly) orthogonal degrees of freedom for locking-aware structural semi-join algorithms: the axis which has to be evaluated (child/descendant/parent/ancestor); the assumed role of the given join input from which the navigational operations start (i.e., the input acts as ancestor/parent or child/descendant); and the index structure to use (document index or element index) for evaluation. A combination of these concepts results in a total number of 12 different structural semi-join algorithms. We use the following naming scheme for the operators: `<axis> + <input> + <index>: {Parent|Ancestor|Child|Desc} {A|B} {Doc|El}`. For example, operator 1 in Figure 7 may be expressed by a `ParentBDoc` operator, because it calculates the parent axis, operates on an input sequence with the role of the child (`B`), and uses the document index for this calculation. Note, for brevity, we only include the information about the index in an operator's name, if the document index is used and omit it, in case of element index usage. For an overview of all possible[6] operators grouped into four classes, refer to Table 1. The row header defines which input for the operator is assumed, whereas the column header defines the output. For clarification of the semantics, each operator is additionally described by an XPath expression where the assumed input element sequence's name is marked in bold face.

Table 1.Classification of semi-join operators

	Output	
Input	ancestor/parent	descendant/child
	Class I: TopFilter	Class II: DownwardStep
parent	//**a**[b]	//**a**/b
	ParentA	ChildA
ancestor	//**a**[.//b]	//**a**//b
	AncestorA	DescendantA
	Class III: UpwardStep	Class IV: BottomFilter
child	//a[**b**]	//a/**b**
	ParentB(Doc)	ChildB(Doc)
descendant	//a[.//**b**]	//a//**b**
	AncestorB(Doc)	DescendantB(Doc)

4.3 Implementation of Selected Operators

The two most challenging problems for the implementation of the twelve operators are a duplicate-free output which is additionally sorted in document order. To guarantee a linear worst-case complexity, the input must not be scanned multiple times (not even partially) and the output order has to be achieved without explicit use of sorting techniques.

4.3.1 Filter Algorithms

The simplest algorithms possible are those, which just act as filters for their input element sequences. For each input element, they check for the existence of a structurally related element with a certain name or predicate, either among relative parent/ancestor elements (BottomFilter) or their child/descendant elements (TopFilter). For an example, consider operator 4 in Figure 7. Its purpose is to filter out those author elements that do not have the value "Ende" as content. Filter algorithms provide a duplicate-free output, whose order is the same as the input order. Essentially, the input is only read, filtered and forwarded, as you can check out by tracing the pseudocode in Figure 9. We process each element at a time using either a document index lookup or an element index lookup. Note, the document index may only be used for the existence check of possible ancestors and parents. In contrast, if we try to find matching elements in the child and descendant axes, we potentially have to navigate large subtrees, which results in prohibitively high access costs. Using the document index or alternatively the element index (line 11) we search for ancestor/parent elements that fulfill the given predicate. Therefore, appropriate (lists of) SPLIDs have to be calculated (lines 6 and 8). Each document index access results in a node lock for the probed element.The element index can be efficiently queried for all four axes in question. Downward queries (child/descendant) result in simple range scans over the corresponding node reference index for *name*. Queries for upward navigation precalculate the list of possible (ancestor/parent) elements and check for their existence in the corresponding node reference index (similarly to lines 5 to 8 in Figure 9). Essentially, downward processing steps result in local axis locks for the queried name, whereas upward queries are protected by node locks.

Input: Seq elements, Axis axis, Pred p, Idx idx
Output: Seq results

```
01 foreach elem in elements do
02
03   // bottom filter
04   if(axis == child or axis == descendant)
05     if(axis == child)
06       Seq s = calculate parent SPLID for elem;
07     else if(axis == descendant)
08      Seq s = calculate ancestor SPLIDs for elem;
09     foreach x in s do
10       // element or document index lookup
11       if(idx.lookup(x, p) is not empty)
12         add elem to results;
13
14   // top filter
15   else
16     // element index lookup
16    if(idx.lookup(elem, axis, p) is not empty)
17       add elem to results;
```

Figure 9. Filter operators

4.3.2 Downward Step Algorithms

In this case, the algorithm processes a sequence of possible ancestors/parents and produces their descendants/children, if they satisfy the given predicate *p*. For the same reason as above, the element index has to be used for this task. The algorithm for the descendant axis (DescendantA) works as follows: For the first incoming element *e*, we use an element index access to return in document order all its descendants that fulfil *p*. Then we compare the next element of the input *e*' with *e*. If *e*' is a descendant of (or equal to) *e* (due to

[6] There are only 12, not 16 different operators, because the option of the index to use is only orthogonal in classes III and IV.

Input: Seq elements as Parents, ElIdx idx, Pred p
Output: Seq results as Children

```
01 Stack stack, postStack;   // for element handling
02 Element la;               // one element lookahead
03
04 foreach elem in elements do
05   la = elem.next();
06  if (stack.isEmpty() or elem.isDescOf(stack.top())
07     placeChildrenFor(elem);
08   else
09     while (not stack.isEmpty() and
10            not elem.isDescOf(stack.top())) do
11       add postStack.pop() to results;
12     placeChildrenFor(elem);
13
14 function placeChildrenFor (Element elem)
15   List children = idx.lookup(elem, Child, p);
16   foreach child in children do
17     if (child < la) add child to result;
18     else add child to Queue q;
```

Figure 10. ChildA downward step operator

element nesting), we may safely skip *e*', because its descendants have already been returned. Otherwise, we proceed with *e*' in the same way as *e*. Because of the ordered input, we can be sure that, if *e*' was not a descendant of *e*, no further descendants of *e* may occur after *e*'. Therefore, the output is also in document order and no duplicates are generated.

The pseudo code of the ChildA operator is depicted in Figure 10. For the child axis, the key issue is the occurrence of nested elements in the input sequence. Suppose, there is a descendant element *e*' following *e* in the input. Then, the children of *e* which fulfil the predicate *p* may partly be located in document order *before* the children of *e*' and partly *after* them. Therefore all elements smaller than *e*' may be directly returned as a result, whereas the others have to be stored for later output (see function `placeChildrenFor()` where *e*' is kept in a look-ahead variable). The rest of the code is responsible for the correct input (using `stack`) and element result handling (using `postStack`). As all algorithms described above, this algorithm is also transformable to meet the ONC protocol, operates in a linear fashion, produces no duplicates, and delivers the correct output order

4.3.3 Upward Step Algorithms

For this class, we can only sketch the ParentB operator. Here, the same problem occurs as for ChildA above, but from the child element sequence's point of view: Several (sibling) child elements may have the same qualifying parent element; therefore, we have to assure that each parent is only returned once. It is not possible to simply memorize the last returned parent and compare the current one for equality, because between sibling elements at level L_1 in the input may—due to the document order—reside other elements which could also return (distinct) parent elements and have a level $L_2 > L_1$. An alternative strategy to scan the set of already identified result elements for duplicate elimination is prohibitively costly.

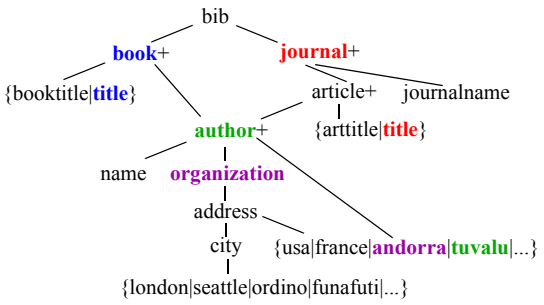
To remedy these problems, we explore the concept of SPLIDs in the following way (see also Figure 11): for the current child element *e*, we use an index lookup to check whether the parent e_p element fulfills predicate *p* (line 5). If so, then each ancestor from the root element to e_p is pushed onto a stack (this is possible, because we can compute all the SPLIDs of these ancestors). Then e_p is marked as an already-found result element and also pushed. As the algorithm proceeds, the stack may grow or shrink. For an element *e*' following *e* in the input, four possibilities occur: 1) e_p' (the parent of *e*') may be equal to the top *t* of the stack ($e_p' = t = e_p$, line 6). Nothing has to be done here. 2) e_p' is a descendant of the top of the stack *t* (line 7). Then the stack grows by pushing all "intermediate" ancestors between *t* and e_p' also on the stack and finally adding a marked e_p'. 3) e_p' is an ancestor *t* (line 9). Now, the stack has to shrink, and the (newly) found e_p' is marked. During the shrinking process, only the marked elements popped from the stack have to be memorized, because they are actual results. This is accomplished with the help of a second stack (`inheritLists`). The exact protocol, how the lists on this stack are handled, can be drawn from the algorithm (lines 23 to 31). 4) e_p' is neither an ancestor nor a descendant of top *t* of the stack, but follows after *t* in document order. Then the stack has to shrink to the least common ancestor between e_p' and *t*, and then grow to e_p.

Input: Seq elements as Children, ElIdx idx
Output: Seq results as Parents

```
01 Stack stack, inheritLists;
02 stack.push(rootSPLID);
03 inheritLists.push(NULL);
04 foreach elem in elements do
05   par = idx.lookup(elem, Parent, p);
06   if(par != null and stack.top() != par)
07     if(par.isDescOf(stack.top())
08       growStack(par);
09     else if (stack.top().isDescOf(par))
10       shrinkStack(par);
11       set mark on stack.top();
12     else if(stack.top() < par)
13       SPLID lca = calcLCA(stack.top(), par);
14       shrinkStack(lca);
15       growStack(par);
16
17 function growStack(Elem elem)
18   push each relative between top and
19     elem onto stack;
20   mark elem and push it;
21  inheritLists.push(NULL); // keep up with stack
22
23 function shrinkStack(Elem elem)
24   while(stack.top().isDescOf(elem)) do
25     if(stack.top() is marked)
26       add stack.top() to Queue q;
27       add Queue from inheritLists.pop() to q;
28     else
29       inheritLists.pop();
30     stack.pop();
31     append q to inheritLists.top();
```

Figure 11. ParentB upward step operator

Finally, the result can be found in its stack `inheritLists` by using the `shrinkStack` function up to the root element of the document. Note, this is the only semi-join operator which is a pipeline breaker, i.e., it has to read all elements before producing any result elements. The problem is intrinsic for this operator and has



TopFilter and DownwardStep Queries:

1) `//book[title]` **or** `//book/title`
2) `//journal[.//title]` **or** `//journal//title`

BottomFilter and UpwardStep Queries:

1) `//author[tuvalu]` **or** `//author/tuvalu`
2) `//organization[.//andorra]` **or** `//organization//andorra`

Figure 12. Structure of a sample document and queries

the following explanation: Suppose, the root element of the document fulfils predicate p and the last element e_L of the input sequence is the last element in the document. Then the root element is matched as the last document but would have been the first one in the result. Because we cannot know this circumstance before accessing e_L, returning the result has to be delayed.

The algorithms for the remaining operators work in a similar fashion and have the same properties as those proposed here. As a conclusion we can state that our operators lead to a very desirable locking situation during query processing, because only small locking granules are required. Therefore, we call them *locking-aware*. Despite this favorable behavior, the query optimizer finally choosing suitable operators for plan assembly must take special care. Using element-at-a-time primitives may become expensive, because for each element lookup at least one page reference is required, if we assume that the higher-level pages of our indexes reside in the database buffer. However, as our performance results reveal, our algorithms provide significant advantages for selective queries (which form a huge class in real-world scenarios).

5. QUANTITATIVE RESULTS

To substantiate our findings, we compared the different algorithms listed in Table 1 in two ways: by one-to-one operator comparison on a single-user system and by comparison in a distributed environment. All tests were run on an Intel XEON computer (four 1.5 GHz CPUs, 2 GB main memory, 300 GB external memory, Java Sun JDK 1.5.0) as the XDBMS server machine and three PCs (1.4 GHz Pentium IV CPU, 512 MB main memory, JDK 1.5.0) as clients, connected via a 100 MBit ethernet cable to the server.

To test the dependency between the run-time performance of our operators and the selectivity of the queries, we generated a collection of synthetic XML documents, whose structure is partly depicted in Figure 12. For each operator (query), we varied the selectivity of the structural join between its input nodes by the following values: 0.01%, 0.05%, 0.1%, 0.5%, and 1%. For example, for the query `//book[title]`, selectivity 0.1% means that 0.1% of all *title* elements have a *book* element as their parent (all others have the *article* element as parent). Additionally, we created 10% "noise" on each input node. In the example, 10% of all *book* elements have the child *booktitle* instead of *title*. On our sample document, we considered a smaller query (Q1) and a larger, more selective one (Q2):

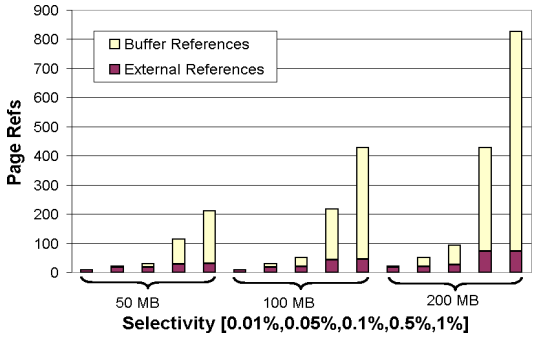


Figure 13. Page References for the ChildB Operator

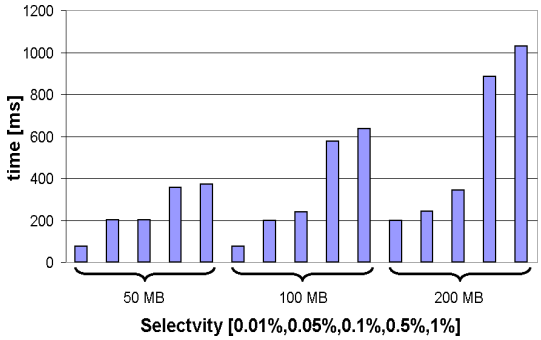


Figure 14. Response time of the ChildB operator

- Q1: `//author[.//funafuti]//name`
- Q2: `//book[.//author//address[.//funafuti][.//andorra]]//title`.

The result size for Q1 varies with the selectivity of the join between *author* and *funafuti,* whereas the result size for Q2 is always 1 (although the selectivities—e.g., between *funafuti* and *address*—vary in the given ranges).

5.1 Single-User Comparison of Operators

To investigate the response time dependency of our operators on the query selectivity and the document size, we measured the number of page references for three document sizes and five selectivities together with the response time of the structural join operators. All measurements were performed with a cold buffer of 250 pages each of 16 KB size, so the buffer size was large enough to avoid page replacement during join processing. In our measurements, we observed for all semi-join operators implemented the same performance characteristics as a function of size and selectivity. Therefore, we can essentially restrict our performance study to a single operator for which we chose the ChildB operator.

The logical page references (consisting of buffer page references plus external page references) show a linear behavior w.r.t. selectivity and document size (see Figure 13), i.e., when the selectivity (document size) doubles, the number of page references also doubles. However, the dominating factor for the runtime perfor-

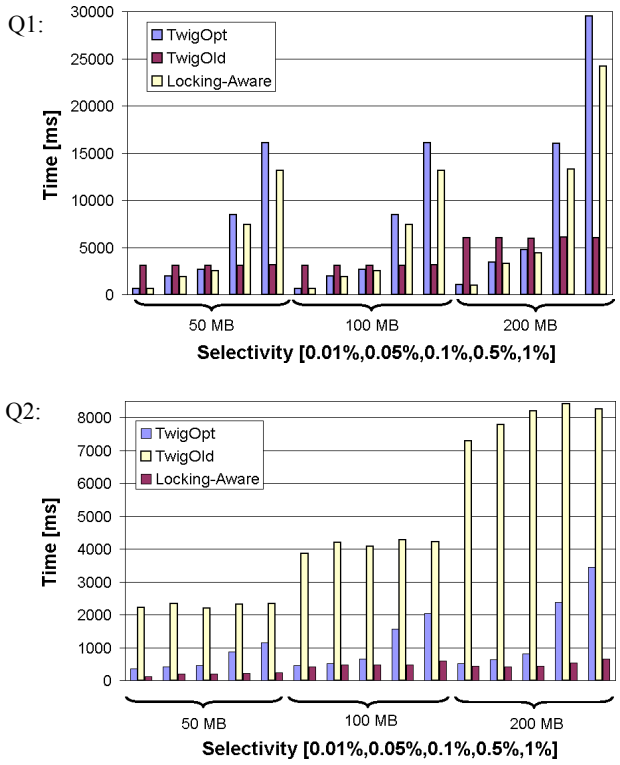


Figure 15. Timing Results of queries Q1 and Q2

mance—the number of external page references—shows a different behavior: When comparing the external page access for selectivity 0.5% and 1%, roughly the same number of pages is transferred into main memory. This effect can be attributed to the page mapping of the accessed elements, i.e., their distribution across external pages, and, in turn, to the increased locality of reference.

The same general behavior can also be observed in Figure 14 which confirms the scalability of our algorithms. On the other hand, the comparison of both types of measurement clearly reveals the dominance of the external page references on the response time. Again, compare the results of 0.05% to 0.1% and 0.5% to 1%, respectively.

To compare the "best" structural join algorithms with our operators, we implemented TwigOptimal (*TwigOpt*) [8] and the original TwigStack algorithm (*TwigOld*) [4] in our XTC system. All single-user tests were run on our server machine, where we picked the best query evaluation plan for our own strategy (*Locking-Aware*) and—to assure fairness—gave *TwigOpt* the necessary *hints*, to pick the next cursor for the best movement.

Figure 15 shows the performance results of queries Q1 and Q2 where the 5 selectivities are grouped together for each document size explored. Running the lower-selectivity query Q1, *Locking-Aware* algorithms are comparable to *TwigOpt* in the entire range and to *TwigOld* up to selectivity 0.1%. For higher selectivity values, *TwigOld* has a striking performance advantage to be easily explained by its sequential access strategy. Because it scans the entire document, it achieves for a given document size roughly constant response times. In contrast, *Locking-Aware* and *TwigOpt* strongly depend on the query selectivity which determines the number of index accesses. Hence in our measurements, selectivity 0.1% embodies the well-known break-even point between sequential access (*TwigStack*) and indexed/random access (*TwigOptimal*[7], *Locking-Aware* operators) [10]. For the very selective query Q2, our approach is clearly the winner on all shown document sizes and (internal) selectivities.

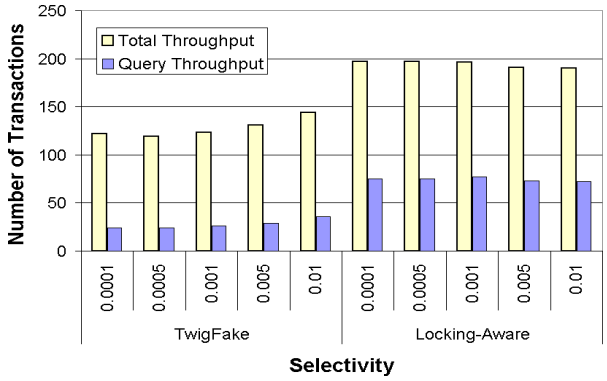


Figure 16. Throughput rates for Q2

5.2 Multi-User Comparison of Operators

Because *TwigOld* scans the entire document to be evaluated, it is obvious that it has to acquire a document lock in multi-user environments. *TwigOpt*, however, proceeds across the document from left to right thereby skipping large ranges of elements whenever cursor calculations reveal that matches are not possible (see Section 3.2). Yet in a multi-user environment, these document ranges—so far not contributing to the current query evaluation—have to be locked, too. Otherwise, a concurrent transaction could insert a subtree in such a range which could contain new matches and therefore could provoke phantoms for the same or a subsequent query of the transaction considered. Hence, these ranges have to be locked resulting in a document lock in most cases. In contrast, *Locking-Aware* operators proceed in subtrees top-down, bottom-up, or in both directions at the same time. Hence, selective locking via axis and node locks becomes possible.

To demonstrate the effective locking behavior of our query evaluation plans, we gave both twig join strategies an advantage: They only have to open the document and access the first element of each defined input cursor, thereby generating a document-wide axis lock (we call this operator the *TwigFake* operator). Note, in our comparison we refer to the best case of a *TwigFake* evaluation, because it does not need to compute any results.

In our experimental setup, each of the three clients runs a certain transaction mix consisting of three long running update transactions that insert new (*name/address/author/...*) elements, and a fourth transaction that executes query Q2 either using the *Locking-Aware* or the *TwigFake* operators, respectively. The results for the different selectivites in the 50 MB document gathered in a two-minute interval are shown in Figure 16. Our locking-aware operators can generate 30% to 60% more total throughput than the *TwigFake* operator. For the throughput of Q2 itself, up three times higher rates can be observed.

[7] TwigOptimal needs indexed access on its *posting lists* (see [8]).

6. CONCLUSIONS

In this paper, we primarily considered the improvement of twig pattern queries—a key requirement for XML query evaluation. For this purpose, we have substantially extended the work on structural joins and holistic twig joins thereby focussing on so far forgotten processing aspects, i.e., optimization of path processing steps in multi-user environments.

While processing twig patterns, our algorithms, supported by appropriate document store and index structures, aim at touching as small data granules as possible and are, therefore, aware of and help to optimize the work of the lock manager. SPLIDs flexibly enable and improve path processing steps in a threefold way:

- They enhance the expressiveness of document and element indexes. For example, they are able to accelerate various navigation steps in the document itself and, in combination with a structural summary, to upgrade element index references to complete path index references.
- They introduce several new degrees of freedom when designing physical operators for path processing steps. This newly gained flexibility considerably increases the solution space for novel algorithms. In particular, they allow the computation and in-memory checking of all axes relationships required for the evaluation of XQuery and XPath expressions.
- They carry path information for all path nodes up to the root which is indispensable for the work of an XDBMS lock manager. Because of the prevalent index access, which is a necessity of all XML language models, jumps to inner tree nodes occur with very high frequency. These accesses have to be isolated by intention locks along the entire ancestor path. This salient feature supported by SPLIDs is missing in all other labeling schemes proposed so far.

We have successfully implemented SPLIDs and concurrency control in our XTC system such that all experiments can be processed under realistic conditions in a complete XDBMS environment.

Performance measurements approved our considerations about locking-aware operators. They are, for a certain query selectivity <0.5%, not only faster in direct comparison, but also lead to substantial throughput increases. Building on our new as well as on existing evaluation algorithms, we are now prepared for the challenge of cost-based query optimization.

REFERENCES

[1] Al-Khalifa, S., Jagadish, H. V., Patel, J. M., Wu, Y., Koudas, N., and Srivastava, D. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE*: 141-152 (2002)

[2] Beyer, K. S., Cochrane, R., Josifovski, V., Kleewein, J., Lapis, G, Lohman, G. M., Lyle, B., Ozcan, F., Pirahesh, H., Seemann, N., Truong, T. C., Van der Linden, B., Vickery, B., and Zhang, C. System RX: One Part Relational, One Part XML. In *Proc. SIGMOD Conference*: 347-358 (2005)

[3] Böhme, T., and Rahm, E. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In *Proc. 3rd DIWeb Workshop*: 70-81 (2004)

[4] Bruno, N., Koudas, N., and Srivastava, D. Holistic twig joins: optimal XML pattern matching. In *Proc. SIGMOD Conference*: 310-321 (2002)

[5] Christophides, V., Plexousakis, D., Scholl, M., and Tourtounis, S. On Labeling Schemes for the Semantic Web. In *Proc. 12th Int. WWW Conference*: 544-555 (2003)

[6] Dewey, M. *Dewey Decimal Classification System.* http://www.mtsu.edu/~vvesper/dewey.html

[7] Cohen, E., Kaplan, H., and Milo, T. Labeling Dynamic XML Trees.In *Proc. PODS Conference*: 271-281 (2002)

[8] Fontoura, M., Josifovski, V., Shekita, E., and Yang, B. Optimizing Cursor Movement in Holistic Twig Joins, In *Proc. 14th CIKM*: 784-791 (2005)

[9] Gottlob, G., Koch, C., Pichler, R. Efficient Algorithms for Processing XPath Queries, In *Proc. VLDB Conference:* 95-106 (2002)

[10] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann (1993)

[11] Härder, T., Haustein, M., Mathis, C., and Wagner, M. Node Labeling Schemes for Dynamic XML Documents Reconsidered, *Data & Knowl. Engineering*, Elsevier (2006)

[12] Haustein, M. *Fine-Ganular Transaction Isolation in Native XML DBS* (in German). Ph.D. Thesis, Univ. Kaiserslautern (2005)

[13] Haustein, M., and Härder, T. Adjustable Transaction Isolation in XML Database Management Systems. In *Proc. 2nd Int. XML Database Symposium*: 173-188, LNCS 3186, Springer (2004)

[14] Haustein, M., Härder, T., Mathis, C., and Wagner, M. DeweyIDs—The Key to Fine-Grained Management of XML Documents. In *Proc. 20th Brasilian Symposium on Databases*: 85-99 (2005)

[15] Jiang, H., Wang, W., Lu, H., and Xu Yu, J. Holistic Twig Joins on Indexed XML Documents. In *Proc. VLDB Conference*: 273-284 (2003)

[16] Jiang, H., Lu, H., and Wang, W. Efficient Processing of XML Twig Queries with OR-Predicates. In *Proc. SIGMOD Conference*: 59-70 (2004)

[17] Mohan, C. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *Proc. VLDB Conference*: 392-405 (1990)

[18] O'Neil, P. E., O'Neil, E. J., Pal, S., Cseri, I., Schaller, G., and Westbury, N. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. SIGMOD Conference*: 903-908 (2004)

[19] Schmidt, A. R., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I., and Busse, R. XMark: A Benchmark for XML Data Management. In *Proc. VLDB Conference*: 974-985 (2002)

[20] Tatarinov, I., Viglas, S., Beyer, K. S., Shanmugasundaram, J., Shekita, E. J., and Zhang, C. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. SIGMOD Conference*: 204-215 (2002)

[21] *W3C Recommendations*. http://www.w3c.org (2004)

[22] Zhang, C., Naughton, J., DeWitt, D., Luo, Q., and Lohmann, G M. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. SIGMOD Conference*: 425-436 (2001)