

# Twig Query Processing Under Concurrent Updates

Christian Mathis, Theo Härder\*  
{mathis | haerder}@informatik.uni-kl.de

University of Kaiserslautern, Germany

## 1. Motivation

An appropriate database language characteristics leading to the success of declarative query processing—and, in turn, to the rise of relational DBMSs in general—always provides more than one way of evaluating a query. This counts for structurally different but logically equivalent query evaluation plans (QEPs) as well as for different implementations of the same logical operator. This principle surely holds for the novel XML database management systems (XDBMSs): Recently proposed operators for XML query processing can be grouped into the logical operators *Structural Join* [1, 22] and *Holistic Twig Join* [3, 6, 16]. Depending on available internal system mechanisms, a lot of opportunities exist how to implement these operators (two of which are presented in this paper).

For the XML query optimizer, it is important to understand the characteristics and particularities of the various operator implementations. Therefore, in the Ph.D. work, we will invent and investigate physical XML query operators and create an “operator toolbox” from which the optimizer can pick the optimal one for QEP assembly. In doing so, we will pay special attention to concurrent XML document modifications in a transactional context. Surprisingly, support for modifications on XML documents has been neglected over the past years, although crucial for the success of XDBMSs as the following four examples illustrate:

1. *XML Messages*. For XML-based electronic interchange, data has to be exported from an internal data format (e.g., RDBMS, file system, repository) into XML. The receiving site, in turn, imports the XML messages into its internal format. Hence, two transformation steps are required. With a full-fledged (i.e., “update supporting”) XDBMS at hand, data storage and transfer could both be XML-based, avoiding the overhead of XML imports/exports. But even if the two sites stick to their internal data representation, an XDBMS can beneficially be used as a message store, enabling transaction-safe insertions of—as well as queries over—XML messages.

2. *XDBMS as data integration vehicle*. Reference [5] introduces a mediator-based XML view (the “Unified View”) integrating heterogeneous data sources for Web manage-

ment. In this scenario, an XDBMS can be used as “integration vehicle” to manage the materialized XML views. Efficient isolation mechanisms guaranteeing high transaction parallelism are of utmost importance, because updates that are propagated to data sources, require more time (implying longer lock requests) than in a non-federated context.

3. *ROX: Relational over XML*. Reference [10] paints the picture of the relational model being mapped onto XML, for a time in the future, where XDBMS become standard and RDBMSs legacy systems. Whether or not this scenario comes true shall not be discussed here; however, in ROX, the XDBMS must—of course—enable concurrent updates.

4. When the relational model does not meet the requirements of the data to be handled (as observed in [9]), the information system designer might choose XML as data model. The XDBMS used not only has to support event-based (SAX), navigational (DOM), and declarative (XPath/XQuery) read access, but also efficient modification operations (as defined by DOM/XUpdate) to compete with the functionality of relational systems.

Therefore, 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. By developing our own native XML database system—the XML Transaction Coordinator (XTC)—, we are able to investigate the strong influence on all system components and their underlying concepts resulting from modification support. Arising important topics are: 1) the design of a *stable XML node identification mechanism* (labelling scheme), that avoids renumbering after arbitrary node/subtree insertions or deletions; 2) the construction of an efficient *physical data model* for the storage and indexing of XML data, requiring low (index) maintenance costs; 3) the embedding of a *locking protocol* that enables maximal transaction concurrency while ensuring isolation offering fine locking granules; 4) the development of a *low-maintenance meta-data catalog* as well as an *adaptive XML query processor* that takes the amount of available main memory as well as the characteristics of the current transaction mix (e.g., the number of read/write transactions, ...) into account; 5) the design and standardization of a *declarative update language*

\* Advisor of this work.

as well as a *distributed XML benchmark environment*, defining a real world scenario with concurrent event-based, navigational, and declarative read/write transaction mixes.

For navigational and event-based access primitives, problems 1 to 3 have been successfully tackled by M. Haustein in XTC [11] (see Sect. 2). For *declarative* access, however, they are subject of ongoing and future research and will be central issues in this Ph.D. work. In this paper, we will elaborate on new approaches concerning physical structural join operators for the XTC query processor.

The rest of this paper is organized as follows: Sect. 2 provides the basic concepts of the XTC (M. Haustein’s work), while Sect. 3 briefly introduces existing strategies for the evaluation of so-called twig queries, as well as our new ideas of structural join operators. Finally, Sect. 4 gives some preliminary empirical results, while Sect. 5 summarizes the related work on the topic discussed.

## 2. XTC—An Overview

XTC adheres to the well-known layered hierarchical architecture: The concepts of the storage system and buffer management could be adopted from relational DBMSs. The access system, however, required new concepts for document storage, indexing, and modification, including locking.

### 2.1 Node IDs, XML Storage, and Indexing

Each XML node in the database has to be uniquely identified by an ID. Well-known identification techniques are Dewey-ID [14], ORDPATH [19], or DLN 2. 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. In general, SPLIDs are immutable, i. e., they support arbitrary node insertions without reassigning IDs of existing nodes; SPLIDs allow the calculation of all ancestor SPLIDs; and finally, given two SPLIDs, all XPath axis relations can be decided without access to the document.

A B\*-tree is used as a *document store* where the SPLIDs in inner B\*-tree nodes serve as fingerposts to the leaf pages. The set of doubly chained leaf pages forms the so-called document container where the XML tree nodes are stored using the format (SPLID, data) in document order. Important for our discussion, the XDBMS creates an *element index* for each XML document. This index consists of a *name directory* with (potentially) all element names occurring in the XML document (Fig. 1). For each specific element name, in turn, a *node-reference index* is maintained which addresses the corresponding elements using their SPLIDs. Note, for the document store and the element index, prefix compression of

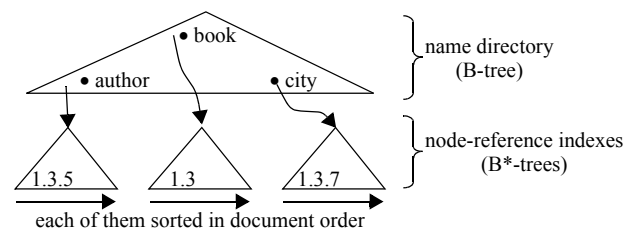


Fig. 1 Organization of the element indexes

SPLID keys is very effective, because both are organized in document order directly reflected by the SPLIDs [14].

### 2.2 XML Node and Axis Locking

Processing XML documents in multi-user environments requires sophisticated isolation mechanisms to perform scanning, navigational, and declarative read operations as well as arbitrary element/subtree insertions/deletions. Due to space restrictions, we only give an overview over the two basic lock types of our hierarchical lock protocol [11]: The so-called *node locks*—having the form (SPLID, lockmode)—provide for fine-grained shared and exclusive access to single XML nodes, their potential attributes and values, as well as access to entire document levels or subtrees. In case of indexed access, e. g., by scanning all *book* nodes via the element index, phantoms have to be avoided, i. e., no further *book* nodes may be inserted by concurrent modification transactions. Therefore, we combine the node-based hierarchical locking with a kind of key-range index locking [18] extended by XPath axis semantics, called *axis locks* in this paper. Axis locks have to form (SPLID, axis, value). For example, the lock (ID, CHILD, title) prevents any concurrent transaction from inserting or deleting a node with value title that resides in the child axis of the node identified by ID. The lock does not restrict parallel transactions more than necessary, because title nodes can be inserted into any other axes areas of the document (e. g., as descendants of ID except children or in disjoint subtrees).

## 3. Twig Query Processing

A twig query is a small tree, whose nodes  $n$  represent simple predicates  $p_n$  on the content (text) or the structure (elements) of a queried XML document, whereas its edges define the desired relationship between the items to match. For twig query matching, the query processor has to find all embed-

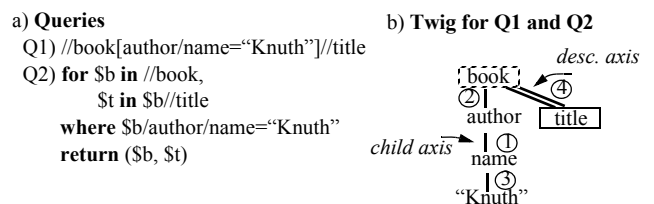


Fig. 2 Sample Queries and their Twig

dings in the queried document, such that each twig node corresponds to an XML item and the twig relationship among the matched items is fulfilled. The result of a twig is represented as an ordered<sup>1</sup> sequence of tuples, where the fields of each tuple correspond to matched items (e.g.  $\langle [book_2, author_5, name_5, \text{“Knuth”}_1, title_7], \dots \rangle$ ).

A large class of twig evaluation methods builds on the *structural join* [1, 22] and the *holistic twig join* [3, 6, 16]. The first approach decomposes the twig into a set of binary structural join operations, each applied to neighbor nodes of the twig (for an example see Fig. 3). In contrast, the holistic twig join algorithms evaluate the twig as a whole via a scan over a set of element sequences (containing only those elements satisfying the node predicates in the twig). We will follow the first approach in this paper.

When running structural and 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, especially to avoid phantoms. All so-far proposed evaluation strategies ignore concurrent (modification) transactions and imply a document-wide axis lock at least for the queried element names and content. Obviously, such an approach has the severe drawback of preventing concurrent document modifications for the queried items. To relieve the blocking situations caused by the lock protocols needed, our new algorithms (see Sect. 3.1) access as small data granules as possible and, therefore, are aware of the lock granules implied.

Another problem of many existing structural join approaches is their need to access XML elements in *document order*, requiring expensive sort operations on intermediate results (as depicted<sup>2</sup> in Fig. 3). Operators processing *unordered* input sequences and, therefore, reducing the number of sort operations, can beneficially be used in these cases. This class of *hash-based* algorithms is sketched in Sect. 3.2.

### 3.1 Upward and Downward Processing Steps

Upward and downward processing steps make intensive use of the document container and the element index, sketched in Sect. 2. The basic idea is to initially scan a small element sequence via the element reference index, but—in contrast to competing join strategies—only for one twig node. For this access, a document-wide axis lock protects the scanned element sequence from concurrent modification. However, be-

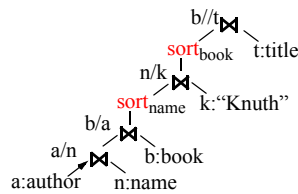


Fig. 3 Sample Plan

cause this sequence is supposed to be small, e.g., the element sequence for a selective predicate, concurrent modifications are slightly hindered. The other nodes are then processed via element-at-a-time lookup in the document or element index, thus acquiring only node locks for upward navigations and local axis locks for downward processing steps, which are sufficient to avoid phantoms.

For example in a transaction  $T_1$ , the twig in Fig. 2 may be evaluated as follows: a) All text elements with the value “Knuth” are scanned via an index access and locked with a document-wide axis lock. Any concurrent transaction  $T_2$  is not able to insert the text value “Knuth” into the document anymore. However,  $T_2$  can still modify elements that participate in  $T_1$ ’s final result, but have not been accessed by  $T_1$  so far (e.g., a *booktitle* element may still be changed to *title*). This is no problem because, if  $T_2$  commits before  $T_1$  reaches any *title* 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 *title* element. b) For each scanned text element, the SPLID of the parent element is calculated by an *upward navigation operation* and accessed via the document index, causing a node lock for this element. Of course, all elements that have another name than *name* are filtered out. In this state, *name* elements may still be inserted, but it is not possible to rename an existing parent of an “Knuth” text element to *name*, because they are protected by node locks. c) Using the same technique as in b), we navigate to the *author* and *book* elements. d) For each *book* element found, a *downward processing step* queries the element reference index for all descendant elements with the name *title*, thus generating a local<sup>3</sup> axis lock for all intermediate *book* elements and the descendant axis. In the final state, no concurrent transaction has the chance to generate a phantom. All elements reached by an upward navigation, such as *author* and *book*, are protected by a node lock, whereas all elements reached via the node reference index in a downward processing step 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.

The access primitives *upward navigation* and *downward processing step* can be implemented as a set of physical structural join operators. Thus, they can easily be embedded into any query processing system that is able to construct query evaluation plans based on structural joins (e.g., [21]). Unfortunately, the use of such element-at-a-time access 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 the quantitative results show, our

<sup>1</sup>In this paper, “ordered” means: sorted in document order from the root to the leaf items.

<sup>2</sup>The arrows in Fig. 3 denote the twig node, the result of the join is ordered by.

<sup>3</sup>In contrast to a document-wide axis lock, a “local” axis lock does not affect the whole document, but only a small part (e.g., a subtree).

algorithms provide substantial advantages for very selective queries (a huge class in real-world scenarios).

### 3.2 Hash-Based Structural Join Operators

Hash-based joins represent another interesting class in our “operator toolbox”. Their advantage is the ability to process unordered and non-indexed input sequences. The basic idea of hash-based join algorithms is to construct a hash table for one of the input sequences A and, then, to sequentially probe the remaining input B. For this purpose, we exploit the expressiveness of SPLIDs to locate possible parents and ancestors in the hash table. Because hashing destroys the order of a possibly sorted input sequence A anyway, we can abandon the sort phase needed by the other classes of structural join algorithms. The output order is determined by the order of B.

As an example, consider the operator between the elements *book* and *title* in Fig. 3: In the first phase (hashing) of the algorithm, each book element (assumed to be a parent) in input sequence A is hashed into hash table HT using the element’s SPLID as the hash key. In the second phase (probing), for each *title* element in input sequence B, the parent SPLID is calculated and probed against HT. If HT contains the parent SPLID of a *title* element, a join match is found.

Because the construction of a hash table is expensive, it is important to use the smaller input sequence for hashing and the larger one for probing (like in traditional hash-join operators). Therefore, our hash-based operators are *symmetric*, enabling the calculation of the same result by hashing either input sequence A or B. Note, regarding locking, hash-based operators resemble the traditional structural join approach in [1], i.e., they do not produce the favorable lock distributions sketched in Sect. 3.1. However, because hash-based algorithms do not rely on indexed or specially ordered input, they can be beneficially used as the in-memory<sup>4</sup> join strategy for partition-based joins when the size of the input sequences is larger than the available main memory.

## 4. Preliminary Quantitative Results

To substantiate our findings, we compared the strategies from Sect. 3.1 by a comparison in a distributed environment. The tests were run on an Intel XEON computer (four 1.5 GHz CPUs, 2 GB main memory, Java Sun JDK 1.5.0) as the DBMS server and three PCs (1.4 GHz Pentium IV CPU, 512 MB main memory, JDK 1.5.0) as clients, connected via 100 MBit ethernet 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 Fig. 4. For the queries, we varied the selectivity of the structural joins by the following values (thus generating different documents):

<sup>4</sup>Partition-based joins create external partitions of the input before applying an in-memory algorithm for join result computation.

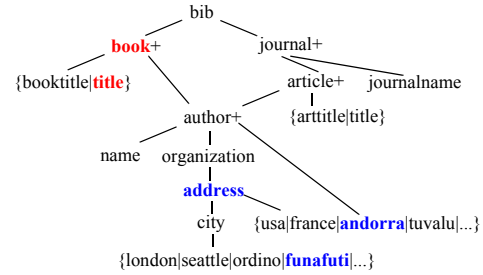


Fig. 4 Structure of a Sample Document

0.01%, 0.05%, 0.1%, 0.5%, and 1%. For example, for the join between *book* and *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 the query Q: `//book[.//author//address[.//funafuti][.//andorra]]//title`. The result size for Q is always 1 (although the selectivities between *book* and *title*, *address* and *funafuti*, and *address* and *andorra* vary in the given ranges).

In our setup, each of the clients runs a certain transaction mix consisting of three concurrent update transactions that insert new (*name/address/author/...*) elements, and a fourth transaction that executes Q either using our approach (*Navi*) or the so-called *TwigFake* operator, which we gave the following semantics: *TwigFake* only has to open the document and access the first element of each defined input cursor, thereby generating a document-wide axis lock. *TwigFake* does not compute any results and can, thus, be used as a best-case representative for all twig join algorithms.

The results for the different selectivities in a 50 MB document gathered in three times two minutes are shown in Fig. 5. Our locking-aware operators generate 30% to 60% more total throughput than the *TwigFake* operator. For Q itself, up three times higher rates can be observed.

## 5. Related Work

Recent publications on concurrency control for XDBMSs can be grouped in proposals targeting node locking [11, 15]

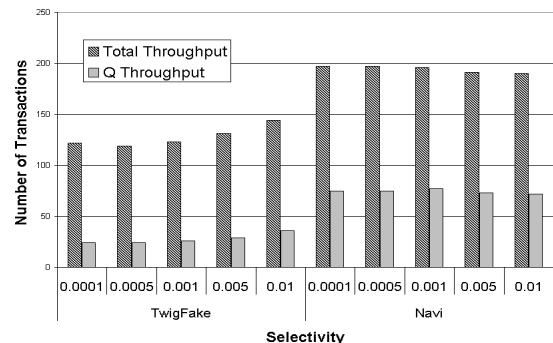


Fig. 5 Throughput Rates for Q

for navigational DOM operations, and path-based (predicate) locking for declarative (XPath) queries [4, 7]. Because a detailed comparison can be found in [12, 13], we only briefly summarize some important aspects: Our upward and downward processing steps try to generate a salutary lock distribution to maximize transaction parallelism. Because they build on node (and axis) locks, it is possible to replace the underlying locking protocols as long as they support a certain common functionality. This idea has been explored as “meta-locking” in [13]. Therefore, our algorithms are independent from the underlying node locking strategy. In contrast, [4, 7] are competing approaches, because they also consider locking for declarative queries. The ideas in [4] can be considered as impractical, because in the proposed *path lock propagation* scheme,  $O(n*m)$  locks<sup>5</sup> are generated for a single read lock request, whereas  $O(n^4)$  comparisons have to be performed in the *lock compatibility* scheme to check the compatibility of two locks. The idea of XMLTM [7] is to employ a DataGuide and predicate locks on the DataGuide’s nodes for concurrency control. However, how the DataGuide is maintained during concurrent updates remains arcane and, furthermore, checking for lock compatibility at the level of meta-data (DataGuide) results in coarser lock granules than checking at the instance level (as in our approach).

As mentioned above, hash-based operators can be applied in partition-based schemes [17, 20], when the size of the input sequence is larger than the available main memory. Competing algorithms for the in-memory join either sort one input sequence or build a heap-based data structure, requiring  $O(n \log_2 n)$  steps, while hash-based operators can process in linear time.

## 6. Conclusions and Outlook

Understanding the characteristics of our as well as competing structural joins approaches collected in the “operator toolbox” is essential for the design of a good query optimizer. Therefore we will run extensive tests to identify those situations, when a specific operator can beneficially be applied.

Furthermore, the generation of a salutary lock distribution is important to provide for high transaction parallelity when concurrent modifications occur. We will explore the concept of *lock prediction* to generate such distributions even for join strategies other than ours.

## References

[1] S. Al-Khalifa et al.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. ICDE: 141-152 (2002)

<sup>5</sup>The size of the document is denoted by  $m$ , whereas the length of the path expression to lock is  $n$ .

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

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

[4] S. Dekeyser, J. Hidders, J. Paredaens: A Transaction Model for XML Databases. In WWW: Internet and Web Information Systems, 7, 29-57 (2004).

[5] M. Flehmig: Data-Driven, XML-Based Web Management in Highly Personalized Environments. Workshop on Information Integration on the Web: 81-88 (2001)

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

[7] T. Grabs, K. Böhm, H-J. Scheck: XMLTM: Efficient Transaction Management for XML Documents. Proc. CIKM (2002)

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

[9] J. Gray: A Call to Arms. ACM Queue 3:3, 30-38 (2005)

[10] A. Halverson, V. Josifovski, G. Lohman, H. Pirahesh, M. Mörschel: ROX: Relational over XML. Proc. VLDB: 264-275 (2004)

[11] M. Haustein, T. Härder: Adjustable Transaction Isolation in XML Database Management Systems, in: Proc. 2nd Int. XML Database Symposium (XSym 2004), Toronto, Canada, LNCS 3186, Springer, 173-188 (2004).

[12] M. Haustein: Feingranulare Transaktionsisolation in nativen XML-Datenbanksystemen, Ph.D. dissertation (2005)

[13] M. Haustein, T. Härder, K. Luttenberger: Contest of XML Lock Protocols. Internal Report: <http://www.dvs.informatik.uni-kl.de/pubs/p2005.html> (2005)

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

[15] S. Helmer, C-C. Kanne, G. Moerkotte: Evaluating Lock-based Protocols for Cooperation on XML Documents. SIGMOD Record, Vol. 33, No. 1, March 2004.

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

[17] Q. Li, B. Moon: Partition Based Path Join Algorithms for XML Data. Proc. DEXA: 160-170 (2003)

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

[19] P. E. O’Neil, E. J. O’Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury: ORDPATHS: Insert-Friendly XML Node Labels. Proc. SIGMOD: 903-908 (2004)

[20] Z. Vagena, M. M. Moro, V. J. Tsotras: Efficient Processing of XML Containment Queries using Partition-Based Schemes. Proc. IDEAS: 161-170 (2004)

[21] Y. Wu, J. M. Patel, H. V. Jagadish: Structural Join Order Selection for XML Query Optimization. Proc. ICDE: 443-454 (2003).

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