Optimistic path-based concurrency control over XML documents

Djamel Berrabah
LIP6 - ANR-05-MMSA-0011
104 av. Président Kennedy,
Paris, France
djamel.berrabah@lip6.fr

Stéphane Gançarski
LIP6 - ANR-05-MMSA-0011
104 av. Président Kennedy,
Paris, France
stephane.gancarski@lip6.fr

Sarah Kaddour Chikh
LIP6 - ANR-05-MMSA-0011
104 av. Président Kennedy,
Paris, France
kaddours@lip6.fr

Cécile Le Pape
LIP6 - ANR-05-MMSA-0011
104 av. Président Kennedy,
Paris, France
cecile.lepape@lip6.fr

ABSTRACT

We present a new approach for concurrency control over XML documents. Unlike most of other approaches, we use an optimistic scheme, since we believe that it is better suited for Web applications. The originality of our solution resides in the fact that we use path expressions associated with operations to detect conflicts between transactions. This makes our approach scalable since conflict detection except in few cases does not depend on the database size nor on the amount of modified fragments. In this paper, we describe and motivate our concurrency mechanism architecture, we describe the conflict detection algorithm which is the core of our proposal and exhibit first experimental results.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*concurrency*

General Terms

Algorithms, Management, Verification

Keywords

XML, XPath, transactions, optimistic concurrency control

1. INTRODUCTION

As the amount of XML data on the World Wide Web is constantly increasing and since many contents on the Web are dynamic, *i.e.* their content can change over time, updating XML becomes crucial for Web applications. For instance, ATOM publishing protocol allows multiple users to concurrently update XML documents called "feeds", while feed readers periodically poll the feed server for new feed entries (XML fragments) in the feed.

Several update languages for XML have been proposed [1]. XUpdate [20] is an XML-based language. It uses XPath to identify a set of nodes, then specifies whether to insert, delete or update these nodes. A set of extensions to XQuery has been proposed by members of the W3C XQuery working group and Patrick Lehti. Variations on these extensions have been implemented and it seems likely that XQuery Update[17] will form the basis of the update syntax in XQuery[1]. With updating facilities, concurrent access to XML documents becomes a more and more important issue in order to improve access performance on XML documents.

It has been shown in [4] that conventional concurrency control methods involved in CVS, relational, and object oriented database systems do not suite XML data well. These methods do not provide a high enough degree of concurrency for XML. Several approaches have been proposed to deal specifically with XML document concurrency control. Most of these solutions are based on locking. Locking based protocols use various types of locks to determine whether a transaction can proceed. Shared locks and exclusive locks are two basic types of locks. A transaction can proceed if the lock on the desired object is compatible with locks held by other transactions on the same object. Locking based protocols can be classified as follow:

- **Xpath-based locking**. Path lock protocols either lock a simplified form of XPath expression [5, 3, 13], use path locks propagation or satisfiability [5, 6].
- **Node-based locking**. Several protocols are based on DOM operations [10, 12, 13]. Different lock modes are defined here such as locks on node children and locks on individual nodes or pointers between them.
- **Graph locking**. A directed acyclic graph locking protocol is proposed in [9]. Locks in this protocol are associated with the nodes in a DataGuide, as in [15].

Due to the pessimistic nature of locking, experiments [11] show that locking overhead may be huge, mainly for applications with few or without conflicts. In relational databases,

[1]Since Fri, 14 Mar 2008, XQuery Update Facility 1.0 is a W3C Candidate Recommendation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSTST 2008 October 27-31, 2008, Cergy-Pontoise, France
Copyright 2008 ACM 978-1-60558-046-3/08/0003 ...$5.00.

optimistic concurrency control (OCC) is an alternative to locking when the conflict rate is low. In optimistic approach, a data is only locked when updates are written in the database (update phase). Therefore, the data can be retrieved and updated by other users at any time other than during this phase. OCC allows the data to be read simultaneously by multiple users and users are blocked less often than its pessimistic locking counterpart [14]. OCC is better suited to Web applications since it does not require locking ressources during the working phase. This is important since, in Web applications, peers are likely to fail and would block other participants if they hold locks.

To the best of our knowledge, there is only one article [16] which proposes two optimistic mechanisms based on snapshot for concurrency control over XML documents: OptiX and SnaX. To perform validation phase, for each transaction T_i, the system keeps track of the set of nodes read, denoted $RS(T_i)$, and the set of nodes written, denoted $WS(T_i)$. In OptiX, a transaction T_i passes validation if for all concurrent transaction T_j that already validated, $WS(T_j) \cap RS(T_i) = \varnothing$. In SnaX, a transaction T_i is validated if for all concurrent transaction T_j that already validated, $WS(T_j) \cap WS(T_i) = \varnothing$.

An important point is that concurrency is not allowed during the validation phase: only one transaction can be validated at a time. Thus, validation phase has to be as quick as possible. In order to improve the validation phase duration, our approach is the following. We assum that both queries and updates are XPath-based. An update statement selects, via Xpath, one or more target nodes for the update, and perform an update operation (insert, delete, update, etc.) on these target nodes. In this context, we aim to reduce the validation phase duration by detecting conflicts based on the transaction code. The set of XPath expressions used for read (resp. write) operations called *ReadPathSet* and denoted $RPS(i)$) (resp. *WritePathSet*, denoted $WPS(i)$) are extracted from the code of each transaction T_i. This extraction is done either at compile time if the transactions are stored procedures, or at runtime during the working phase for on-demand transactions. During the validation phase, ReadPathSets and WritePathSets are compared to detect conflicts. The first advantage of this approach is that no additional work is done during the read phase. More important, in most cases conflict detection is not dependent on the size of the database nor on the amount of modified fragments. On the opposite, in the approach of [16], the ReadSets and WriteSets may become very large if the Xpath expressions are not narrow. The drawback of this code-based conflict detection is that not every conflict can be detected from only XPath expressions comparison, resulting in false positive conflict detections. We thus propose several ideas to reduce the number of false positive conflict detection, for instance by relying on the XML Schema whenever possible or by taking into account the semantics of operations.

The paper is organized as follows. Section 2 presents preliminary XML models for transaction management. Section 3 shows the global architecture of our concurrency control mecanism. Section 4 presents our path-based conflict-detection method. In section 5 we present implementation results to estimate the efficiency of our approach. Section 6 concludes.

2. TRANSACTION MANAGEMENT IN XML DOCUMENTS

This section presents the SchemaGuide associated with a document, the access and transaction models and a running example to show how XML data is accessed by transactions. It also describes globally how our mechanism works.

2.1 XML Schema and SchemaGuide

Let us consider an example of XML document on Fig 1. We assume that documents are associated with a document type, here an XML schema.

XML Schema [18] is an XML-based alternative to DTDs which is more powerful than DTDs. It describes and restricts the content of XML documents. Due to lack of space, the XML Schema, associated to the XML document in Figure 1, is not given. Instead, we show the Schema Guide associated to the document where every path in the document has exactly one path in the Schema Guide. A Schema Guide is a simplified representation of the XML Schema, which sums up the structural relationships between nodes. It is based on the DataGuides [8] definition and is obtained by transforming the XML Schema.

2.2 Access model

Since most of XML languages are based on XPath, document data are accessed using XPath expressions. An access to a document is an operation over this document to retrieve, add, modify or remove a fragment which can be a node or a sub-tree. We distinguish two kinds of operations:

- *Read-operation*: a read-operation is a query, noted *select*(p), on the document D where p is an XPath expression. The result of such an operation is composed of all the fragments located by the XPath expression.

- *Write-operation*: a write-operation changes the content of the document D. The possible write-operations are *append, insert-before, insert-after, update* and *delete.*

  - *append(f,p)* This operation adds the fragment f as a descendent of the node(s) located by the XPath expression p.
  - *insert(f BEFORE p)* This operation inserts the fragment f such as its root is the preceding sibling of each node matching p.
  - *insert(f AFTER p)* This operation inserts the fragment f such as its root is the following sibling of each node matching p.
  - *update(n,p)* This operation replaces the node(s) located by the XPath expression p by the new node n.
  - *delete(p)* This operation removes the node(s) located by the XPath expression p.

More complex write operations, such as replace or move, can be defined by combining the preceding operations.

2.3 Transaction model

Let $T_1, T_2, ..., T_m$ be transactions. A transaction is a sequence of operations noted $T_i(O_1, O_2, ..., O_n)$ where $O_{i,i=1,n}$ is either *select, append, insert-before, insert-after, update* or *delete.* As shown above, each operation O_i is associated with

XML Document

```
<company>
   <employee id="em65">
      <lname>Dubois</lname>
      <fname>Christian</fname>
      <address>
         <number>8</number>
         <street>Skolem</street>
         <city>Paris</city>
      </address>
      <child age="12" size="1.65">
         <fname>Julien</fname>
         <school>
            <name>Jean-Moulin</name>
            <address>
               <street>Gabriel Peri</street>
               <city>Saint Denis</city>
            </address>
         </school>
      </child>
      <salary>2000</salary>
   </employee>
   <employee id="em45">
      <lname>Quimousse</lname>
      <fname>Pierre</fname>
      <address>
         <street>Oscar Wilde</street>
         <city>Juvisy</city>
      </address>
      <child age="17" size="1.80">
         <fname>Jean</fname>
         <fname>Pierre</fname>
         <fname>Simba</fname>
      </child>
      <salary>2100</salary>
   </employee>
</company>
```

Associated Schema Guide

```
<company>
   <employee id=""><lname/><fname/>
      <address><number/><street/><city/>
      </address>
      <child age="" size=""><lname/><fname/>
         <school>
            <name/>
            <address>
               <number/>
               <street/>
               <city/>
            </address>
         </school>
      </child>
      <salary/>
   </employee>
</company>
```

Figure 1: An XML document and its Schema Guide

an XPath expression p to locate the set of nodes to be accessed. When a transaction has executed all its operations, it asks for validation and enters the validation phase.

In the following, we consider the transactions defined below. For sake of simplicity, we only consider transactions composed of a single operation.

T_1 : $update(<fname>\ Alain\ </fname>\ ),//fname[1]$

T_2 : $delete(/company/employee/address/number)$

T_3 : $insert(<fname>Henri</fname>\ AFTER\ //employee[lname =' Dubois']/fname))$

T_4 : $insert(<fname>Jean</fname>\ BEFORE\ //employee[lname =' Dubois']/fname)$

T_5 : $update(<fname>\ Louis\ </fname>, /company/employee/child[@age = 12]/fname[. =' Julien'])$

T_6 : $update(<fname>\ John\ </fname>, /company/employee/child[@size = 1.62]/fname[. =' Julien'])$

T_7 : $insert(<fname>Henri</fname>\ AFTER\ //employee[lname =' Dubois']/lname)$

T_8 : $insert(<Age>45</Age>)\ AFTER\ //employee[lname =' Dubois']/fname[1])$

2.4 Optimistic concurrency control

In our concurrency control mechanism, a transaction T_i follows three phases: a *working phase*, a *validation phase* and an *update phase*.

1. In the working phase, the read-operations are performed on the document and the changes carried out by the write-actions are stored in a temporary space dedicated to this transaction. During this phase, the RPS , the WPS (*cf.* Section 1) and the operation semantics are extracted from the transaction code or retrieved if this was done during compile time.

2. When the transaction completes, it requests its validation phase. Throughout this phase, a check for conflicts between the transaction T_i and the concurrent ones is performed. Validation phase of a transaction T_i is successful if for all concurrent transactions T_j already validated, there is no conflict between their respective RPS and WPS. This ensure the serializability of concurrent executions.

3. If the validation phase is successful, the changes carried out by the transaction become permanent in the document (update phase), otherwise the transaction is aborted, its temporary space freed and it will be restarted.

3. GLOBAL ARCHITECTURE

Figure 2 shows the internal architecture of our mechanism. It relies on the following modules:

- The **Diff Processor** is based on hierarchical change detection algorithms [2]. It compares the old and new version of an XML document and stores differences in a delta file. The delta file format is based on a description language such as XUL [19] and XUpdate [20]

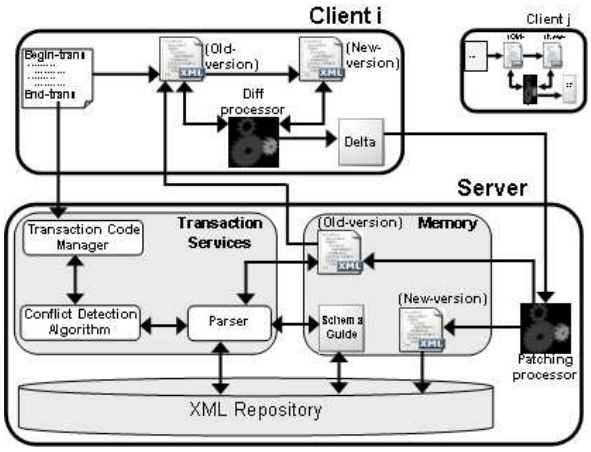

Figure 2: XML OCC architecture overview

- The **Transaction Code Manager** gets the transaction code, extracts path expressions (RPS and WPS) and combines them with path expressions of the concurrent transactions.
- The **Conflict Detection Algorithm** detects conflicts between operations relying on their path expressions given by the Transaction Code Manager.
- The **Parser** allows to load a document and generate its Schema Guide (if does not exist). It may also be called by the conflict detection algorithm if an access to the Schema Guide or to the document is necessary.
- The **Patching Processor** receives the deltas from the diff processor and applies them to the documents to make changes visible to subsequent transactions.

During the working phase, the client asks for an XML document (old-version) on which its transaction is performed. The Parser loads this document into memory, generates its Schema Guide from its XML Schema (if not already done) and saves it in the XML repository. A copy of the XML document is sent to the client (new-version), and all subsequent updates are directed to this copy. During the validation phase, the Conflict Detection Algorithm checks conflicts between the transaction path expressions, *i.e.* if the path expressions lead to the same fragments in the XML document. If necessary, the Parser uses either the Schema Guide or the XML document to continue checking conflicts according to the possible cases (see section 4.1). If the validation succeeds, then the transaction enters the update phase. The Patching Processor receives the delta file from the Diff Processor and applies it to the document to make changes of the transaction global in the document.

4. CONFLICT DETECTION

This section presents how we manage the validation phase. Our method is based on a path-based conflict detection algorithm presented in Section 4.2. This algorithm detects *potential conflicts* between operations, based on their path expressions.There is a potential conflict between two operations if, when evaluating their path expressions, there *may* have at least one node in common. As potential conflicts include actual conflicts, using potential conflict ensure that the algorithm is correct: it detects all the actual conflicts. However, in some cases, potential conflict may not correspond to actual conflicts, as it was the case with a preliminary version of this algorithm introduced in [7]. This would lead to unnecessary transaction aborts, which raises a performance problem. We first show in Section 4.1 the main difficult issues raised by accurately detecting potential conflicts, in order to minimize the number of unnecessary aborts. Then, in Section 4.2, we present the detection algorithm and show how it solves the above mentionned issues.

4.1 Main issues

There are mainly three cases where it is difficult to decide whether two XPath based operations are in conflict or not. Each of them requires a different solution, as explained in the following.

4.1.1 Operations semantics

Two write operations can be considered in conflict whereas they are not, if we do not take into account the semantics of the operations, but only the path expression they are based on. Two operations, with conflicting paths, may be commmutative and thus not conflicting. This depends on the operation type. For instance, in Section 2.3., T_3 and T_4 are defined over the same path expression, thus would be conflicting according to their path expression. However, they are not conflicting since one inserts a sibling on the left of the target node, while the other inserts a sibling at the right of the target node.

On the other end, two operations can be in conflict even if their path expressions are not. For instance, in T_7 and T_8 path expressions are not in conflict, but T_8 insert a node *age* after the first *fname* of the employee *'Dubois'* while the T_7 insert a node *fname* as the first *fname* for the same employee. Thus, T_7 and T_8 are conflicting.

Those cases are detected in our algorithm by the inConflict() function (see section 4.2).

4.1.2 XPath wildcards and document types

As XPath expressions may contain wildcards such as '//' and '*', it may be not possible to decide if two expressions are in conflict, just by considering their path.

Consider transactions T_1 and T_2 of Section 2.3. If T_1 and T_2 execute concurrently on the example document, they are not conflicting. However, they may be conflicting with respect to another document, because '//' may represent any path. Thus the algorithm would return a potential conflict. Fortunately, the problem can be solved in most cases if documents are typed. Indeed, documents types specify which elements can contain which other elements, which elements are optional or required, and which elements contain data. These specifications allow us to check expressions which are not developed, *i.e.* those containing '*' or '//'. For instance, we can infer that there is no conflict between T_1 and T_2 by checking in the document type that *number* is a leaf node, thus can not be an ancestor of *fname*.

In our approach, we assume that documents are typed through a corresponding XML Schema. For sake of performances, the schema is transformed into a Schema Guide

which contains all the possible paths a document may have with respect to the schema. Functions isAncestor() and isFather() use the SchemaGuide to solve cases with wildcards, respectively '//' and '*' (see section 4.2).

4.1.3 Undecidability

Some times, and because we are using static information, exact conflict detection is not decidable, even if using document types.

Consider the transactions T_5 and T_6 defined in Section 2.3. There is no way to determine if the two transactions actually conflict, since there may or not exist a node with both $@age = 12$ and $@size = 1.62$, but this only depends on the current document state. In this case, we propose to extend our path conflict detection by testing if a real conflict happened. This can be achieved in two different ways and depends whether nodes are identified or not. If nodes are identified, then it is sufficient to test if the two possibly conflicting path expressions *actually* touch nodes in common. This can be implemented by maintaining an index on the modified nodes, or by labbeling nodes with transaction identifiers, as in [16]. However, this implies an overhead due to nodes identification for all documents. To avoid this overhead, a simple solution is to evaluate path expressions, by combining path expressions of the possibly conflicting operation to check whether there is actually a conflict between the two path expressions. In the case of T_5 and T_6, we send the query $/company/employee/child[@age = 12][@size = 1.62]$ on the last committed version of the document, and conclude that there is a conflict between T_5 and T_6 if and only if the query returns a non-empty set of nodes. This is achieved in our algorith by the function exists() (see Section 4.2).

4.2 Path-based conflict detection

This Section presents our conflict detection algorithm. The algorithm takes as input two concurrent operations and returns *true* if a potential conflict exists.

Let x and y be operations. We note X the path expression of x. X.op is the operation semantics. In the current version of the algorithm we take into account two kinds of operation semantics: insert-before (resp. insert-after), denoted BEFORE (resp. AFTER).

The main function inConflict()[2] decompose binary expressions (*e.g.* X = /a/b UNION /a/c) if they exist, into simple expression (*e.g.* /a/b and /a/c), depending to the semantic of operations (X.op), calls inConflictSExp() and isPrecedingSibling() functions, if necessary, and returns *false* if the two operations are not in conflict, *true* otherwise[2].

The inConflictSExp() function recursively analyzes the path steps of both expressions, starting from the first step. When there is a possible conflict, the function returns true; otherwise, *i.e.* there is no conflict for sure, it returns false. Note that, each time there is a RETURN instruction, the execution terminates: each condition is tested if the preceding tests failed. The main components of the inConflictSExp() function are explained in the following points:

- X.first refers to the first path step of X without '/', if it exists, and X.rest refers to the remaining steps (*e.g.* X=/a/b/c, X.first=a, X.rest=/b/c). Node, Attribute and Leaf represent, respectively, an element, an attribute and the other node types (processing instruction(), text(), comment).

```
boolean inConflictSExp(XPath_exp X, XPath_exp Y) {
1. If (X begins with'/' and Y begins with'//')
   return Parser.isAncestor(X, Y.first);
   If (X begins with '//' and Y begins with '/')
   return Parser.isAncestor(Y, X.first);
   If (X begins with '//' and Y begins with '//') {
    If (X.first==Y.first) return (inConflictSExp(X.rest,Y.rest);
    return ((Parser.isAncestor(X, Y.first)
    or Parser.isAncestor(Y, X.first)); }
2. If(X.first and Y.first are both Node,Attribute,Leaf or '*') {
    If (X.first != Y.first) return false;
    If (X.rest==null or Y.rest==null)return true;
    If (X.second and Y.second are both Predicate) {
     If(attribute names are different){
      If(Parser.exist(X[X.second][Y.second]==false)
      return false;
     }else if(DisjunctPredicate(X.second,Y.second)==true)
     return false; }
    return inConflictSExp(X.rest,Y.rest); }
3. If (X begins with '*' and Y.first is Node) {
    If (X.rest==null )return true;
    If(Parser.isFather(Y.first, X.second)==false)
    return false;
    return inConflictSExp(X.rest, Y.rest); }
4. If (X.first is Node and Y begins with '*' )
   return inConflictSExp(Y,X);
5. If (X.first=='@*' and Y.first is Attribute) {
    If (X.rest==null )return true;
    return inConflictSExp(X.rest, Y.rest); }
6. If (X.first=='@*' and (Y.first is Leaf or Node))
   return False
}
```

Figure 3: Conflict detection for XPath expressions X and Y.

- Function DisjointPredicates() verifies if the first steps of two expressions have disjoint predicates, *i.e.* returns true if both have predicates and these are disjoint, false otherwise[2].

- Function exist() is called when the algorithm deals with two path expressions containing predicates with different attribute names. This function combines this predicates in a single expression and queries the document. If the result of the query is a non-empty set of nodes, then this function returns that there is a conflict[2].

- Function isAncestor() (resp. isFather()) verifies, for two expressions X and Y with Y contains '//' (resp. '*'), if the last step of X is an ancestor (resp. a father) of the step following '//' (resp. '*') in Y[2].

5. IMPLEMENTATION AND EVALUATION

We implemented two main part of the architecture presented in Section 3: the Conflict Detection Algorithm and the Parser. Both are implemented in Java. Experiments were led on a PC with two 1.66 GHz processors and 2GBytes of core memory, under Microsoft Windows Vista and Linux operating systems.

The Parser is devoted to two tasks: loading the XML document using JDOM and generating the Schema Guide, if does not exists, using XSLT. We measured the time for each task, on the example document. Those tasks where

[2]Due to lack of space, InConflict(), IsPrecedingSibling, IsFather, Exist() and DisjointPredicates() functions are not presented here.

performed in approximately respectively 100 ms. and 200 ms. Those figures are reasonable if we keep in mind that a document is loaded only if it is not present in the server cache memory.

Performances of the detection algorithm are more challenging, since it is the core of our approach and is used for each couple of operations of concurrent transactions. We ran several experiments, with different operations types : simple types, that do not require access to the Schema Guide, complex types containing either '//' or '*' and undecidable types that require access to the document.

For the first operation type, the decision time is always much less than 1 ms. For the the second operation type, the decision time is in the order of some (between 1 and 4) ms. For the third operation type, the time of checking depends on the document size.

Those results are very encouraging since the optimistic approach requires a very fast validation phase. Furthermore, the transaction code can be sent to the detection algorithm during the working phase, so that the algorithm, if idle, can start to check for conflicts in parallel with transaction execution on the client, which even speeds up the validation phase.

6. CONCLUSION AND FUTURE WORK

This paper presents a new approach for concurrency control over XML documents. We defined the context of our work: transactions for XML and optimistic concurrency control. Then, we showed how we manage the validation phase by detecting potential conflicts between operation, based on the XPath expressions that locate nodes manipulated by operations. Our solution is based on a conflict detection algorithm and we described the main issues we solved so that the algorithm detects more conflicts than the previous version introduced in [7]. On opposite to most of the existing approaches that use locking, we use an optimistic concurrency control scheme. As mentioned in the introduction, this scheme is better suited to Web applications where few conflicts happen because most of the transactions are read-only and because the probability of a client to fail and block other participants if it holds locks is higher.

As far as we know, [16] is the only other approach for optimistic concurrency control over XML data. The main difference with us is that they perform conflict detection at the node level, while we detect potential conflicts at the path expression level. This is more efficient mainly when the database is large (our approach does not depends on the database size) and/or when transactions manipulate big amounts of data. Indeed, in this case, node-based conflict detection has to compare big sets of updated nodes to conclude whether there is a conflict or not.

We implemented a first version of our system and performed experimental validation of our ideas. As shown in section 5, our results are very promising. However, we must run our system over different workloads and/or benchmarks, and compare its results with other approaches. We are curently investigating the two following issues: (1) take more semantics into account, for instance when one operation deletes a subset of the set of nodes that the other operation deletes, and (2), have a very *fast update phase*. To this end, we will choose the best Diff and Patch algorithms and implement the Diff and Patching Processors (see section 3).

7. REFERENCES

[1] R. Bourret. Xml and databases. In *Internet report*, 2005. Available at http://www.rpbourret.com/xml/XMLAndDatabases.htm.

[2] S. S. Chawathe. Comparing hierarchical data in external memory. In *25th Very Large Data Base Conference (VLDB)*, pages 90–101, Edinburgh, Scotland,UK, 1999.

[3] E.-H. Choi and T. Kanai. Xpath-based concurrency control for xml data. In *the 14th Data Engineering Workshop*, page 302?313, Ishikawa, Japan, 2003.

[4] S. Dekeyser and J. Hidders. Path locks for xml document collaboration. In *3rd Conference on Web Information Systems Engineering (WISE)*, pages 105–114, Singapore, 2002.

[5] S. Dekeyser and J. Hidders. A commit scheduler for xml databases. In *The fifth Asian-Pacific Web Conference*, pages 83–88, Xian, China, 2003.

[6] S. Dekeyser, J. Hidders, and J. Paredaens. A transaction model for xml databases. *World Wide Web journal (WWW)*, 7(2):29–57, 2004.

[7] S. Gancarski, C. L. Pape, and A. L. Gancarski. Freshness control of xml documents for query load balancing. In *Proc. of Xantec'07 (DEXA workshops)*, pages 35–39. IEEE Computer Society, 2007.

[8] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *23th Very Large Data Base Conference (VLDB)*, pages 436–445, Athens, Greece, 1997.

[9] T. Grabs, K. Böhm, and H.-J. Schek. Xmltm: Efficient transaction management for xml documents. In *ACM International Conference on Information and Knowledge Management (CIKM)*, pages 142–152, McLean, VA, USA, 2002.

[10] M. Haustein and T. Härder. Fine-grained management of natively stored xml documents. In *Internal Report*, 2004. Available at: http://131.246.18.10/pubs/papers/HH04.Int-Report.html.

[11] M. Haustein, T. Härder, and K. Luttenberger. Contest of xml lock protocols. In *32nd international conference on Very large data bases(VLDB)*, pages 12–15, Seoul, Korea, 2006.

[12] S. Helmer, C. Kanne, and G. Moerkotte. Evaluating lock-based protocols for cooperation on xml documents. *ACM SIGMOD Record*, 33(1):58–63, 2004.

[13] K.-F. Jea, S. Chen, and S. Wang. Locked-based concurrency control for xml document models. In *The International Computer Symposium*, pages 165–172, Taiwan, 2002.

[14] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 9(4):213–226, 1981.

[15] P. Pleshachkov, P. Chardin, and S. O. Kuznetsov. Xdgl: Xpath-based concurrency control protocol for xml data. In M. Jackson, D. Nelson, and S. Stirk, editors, *BNCOD*, volume 3567 of *Lecture Notes in Computer Science*, pages 145–154. Springer, 2005.

[16] Z. Sardar and B. Kemme. Don't be a pessimist: Use snapshot based concurrency control for xml. In *The 22nd International Conference on Data Engineering(ICDE)*, page 130, Atlanta, GA, USA,

2006. poster paper.

[17] Xquery update facility 1.0. http://www.w3.org/TR/xquery-update-10, August 2008.

[18] Xml schema. http://www.w3.org/XML/Schema, October 2004.

[19] Introduction to xul. http://developer.mozilla.org/en/docs/Introduction_to_XUL, January 2005.

[20] Xupdate-xml update language. http://xmldb-org.sourceforge.net/xupdate/, November 2000.