Control and Cybernetics

vol. **38** (2009) No. 1

SXCCP+: Simple XML Concurrency Control Protocol for XML Database Systems*

by

Krzysztof Jankiewicz

Institute of Control and Information Engineering
Poznan University of Technology
Piotrowo 2, 60-965 Poznan, Poland
e-mail: Krzysztof.Jankiewicz@cs.put.poznan.pl

Abstract: Increasing significance and popularity of XML is the main reason why many commercial object-relational database management systems (ORDBMSs) developed XML storage and processing functionality. Additionally, there are new types of specialized database systems known as 'native' XML database systems. As we know, concurrency control is one of the most important mechanisms in DBMSs. Unfortunately, concurrency control mechanisms, which have been used so far in commercially available native XML DBMSs, offer very low degree of concurrency. The development of universal and effective concurrency control protocol for accessing XML data, with high degree of concurrency, is a necessary condition of the native XML database growth.

The aim of this paper is to propose a new concurrency control protocol in XML document access. This protocol is based on primitive operations, which can be treated as a unification platform for any of the XML access methods.

Keywords: concurrency control, XML, databases, protocols.

1. Introduction

Currently, XML document processing is one of the major areas in data processing technology. The popularity of XML is a result of its simplicity and elasticity. Due to these features, XML is the main standard in the complex, variable and semi structured data exchange. Increasing significance and popularity of XML is the main reason why many commercial object-oriented-relational DBMSs developed XML storage and processing functionality. Additionally, there are new types of specialized database systems known as 'native' XML database systems.

*Submitted: June 2008; Accepted: October 2008.

Concurrent and uncontrolled access to XML database systems, like in relational and object-oriented database systems, may lead to data inconsistency. Concurrency control problem was widely considered in the literature. A variety of correctness criteria for concurrency control in database systems and a variety of concurrency control protocols were proposed. Conflict serializability is the main, commonly accepted concurrency control criterion. Developed protocols represent three main approaches: locking, time stamp ordering and optimistic. Mechanisms used so far in commercially available native XML DBMS are based on locking protocols and offer very low degree of concurrency and, thus, very low processing performance. There is an obvious need to develop new methods of concurrency control in XML database systems, which provide database consistency and acceptable degree of concurrency. These methods should provide acceptable performance, taking into account specific XML document features and variety of modification methods. There have been few propositions to solve this problem so far (Helmer, Kanne and Moerkotte, 2004; Haustein and Härder, 2003; Grabs, Böhm and Schek, 2002; Dekeyser and Hidders, 2002; Choi and Kanai, 2003; Jea, Chen and Wang, 2002; Pleshachkov, Chardin and Kuznetsov, 2005). These solutions differ as to the degree of concurrency and assumed access methods to XML documents.

In Jankiewicz (2006) we presented a brief survey of proposed concurrency control protocols and their critical analysis. The protocol classification was also presented. The criteria of evaluation were defined and protocol analysis was made on the basis of proposed criteria. Concurrency control protocols which were analyzed in Jankiewicz (2006) can be classified in several ways. For example, according to the assumed access method to XML documents, we can distinguish the following classes of protocols: based on DOM API and based on XPath standard. There is no protocol unassigned to the particular access method. The aim of this paper is to propose a new concurrency control protocol for XML database systems, which is independent of the access method. We have to stress that different access methods coexist in various 'native' XML databases. They are used depending on users preferences and needs. Due to this fact the independence of concurrency control protocols from access method has the key meaning for future development of XML database systems. Concurrency control mechanisms used so far in commercially available 'native' XML DBMS are based on locking protocols and offer very low degree of concurrency and, thus, very low processing performance. There is an obvious need to develop new methods of concurrency control for XML database systems, which provide database consistency and acceptable degree of concurrency. These methods should provide acceptable performance, taking into account specific XML document features and a variety of modification methods.

The structure of this article is as follows. In Section 2 main XML processing standards are presented. Section 3 presents the proposal of a new concurrency control protocol. Section 5 concentrates on conducted experiments and analysis of the results. Section 6 presents conclusions and directions for future work.

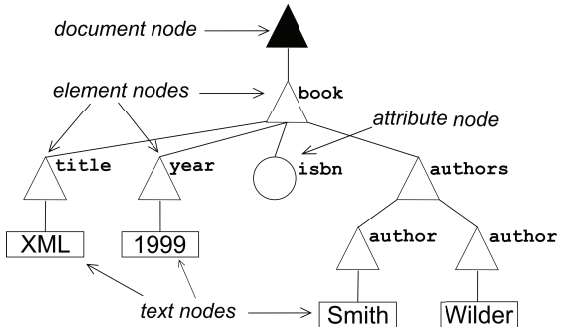

Figure 1. Example of DOM tree

2. XML Interfaces

Extensible Markup Language (XML) (Clark and DeRose, 1999) is a simple text format based on markups, derived from SGML (ISO 8879:1986). XML, originally designed as an electronic standard for distribution of information, was widely used as a data exchange format. An example of XML document is presented in Fig. 2. The sample document is composed of one root element `book`, which consists of three inner elements: `title`, `year`, `authors`. The element `authors` consists of two `author` elements. Additionally, the `book` element has an attribute named `isbn`. The content of the attribute, as well as of the four elements (`title`, `year`, `author`), is a simple string–text node.

There are many standards for processing XML documents. DOM API and XPath are the most important among them.

DOM (Document Object Model) API standard is one of the oldest standards related to XML document access. It provides a set of objects to represent XML documents and the interface to documents access and modifications. It is independent of programming language. Because of DOM API, we can remap XML documents to their object representation, read and modify their content, navigate through their structure and modify them. DOM representation of an XML document is similar to a tree structure, and, therefore, we call it DOM tree. An example of DOM tree, which represents the XML document from Fig. 2 is given in Fig. 1.

In the DOM API interface we can distinguish between procedures and functions to read XML document and procedures and functions to modify it.

The most important procedures and functions to read the structure of XML document are: `firstChild` – reads first child element of a node, `lastChild` – reads the last child element, `previousSibling` – reads previous sibling of a node, `nextSibling` – reads next sibling of a node, `getNodeById` – reads node with determined value of `ID` attribute, `getElementByTagname` – reads nodes with determined tag name. The main methods to read the content of XML

Table 1. Examples of XPath expressions

XPath expression (and its short form)	Addressed XML document fragments
`/child::book/attribute::isbn` (`/book/@isbn`)	`KD-12345-XY`
`/descendant-or-self::author` (`//author`)	`<autor>Smith</author>` `<autor>Wilder</author>`
`/child::book/child::year/child::text()` (`/book/year/text()`)	`1999`
`/child::book[child::year='1999']/_ descendant-or-self::autor[1]` (`/book[year='1999']//author[1]`)	`<autor>Smith</author>`

document are: `nodeValue` – reads content of the text node, `nodeName` – reads tag value of an element, `getAttribute` – reads attribute with determined name.

On the other hand, we have procedures and functions to modify the structure of XML documents: `insertBefore` – inserts new node before the current node, `replaceChild` – replaces current node with new a node, `removeChild` – removes node, `appendChild` – inserts new node as the last node of child nodes. We have methods to modify content, too: `appendData` – inserts text at the end of text node, `deleteData` – deletes text node fragment, `insertData` – inserts text node into current element, `replaceData` – replaces text in the node, `setAttribute` – sets attribute value.

DOM API standard is a procedure interface, which means that user, which uses it, defines modification or read operation in a procedural way. A different approach to define modifications and reads is a rule-oriented approach. There are many standards related to XML that use rule-oriented approach. One of the most important of them is XPath.

XPath can be used to address XML document fragments. It uses XPath expressions – locations path, which are similar to path expression used in file systems. Each XPath locations path consists of one or more location steps, each separated by a slash '/' symbol. Each location step consists of: (1) optional axis specifier, which is used to set direction of tree navigation, (2) node test, which is used to do some node tests dependent on the axis specifier, and (3) optional predicate, which is a kind of condition of the additional selection of nodes. The syntax for a location step is as follows:

`axisname` :: `nodetest`[`predicate`]

XPath expressions can use regular expressions, functions etc. Table 1 contains some examples of XPath expressions that can be used to address the XML document given in Fig. 2.

XPath is widely used in XML documents processing. For example, it is a base for XML document transformation standards like XSLT (Extensible

```
<book isbn='KD-12345-XY'>
  <title>XML</title>
  <year>1999</year>
  <authors>
    <author>Smith</author>
    <author>Wilder</author>
  </authors>
</book>
```

Figure 2. XML document

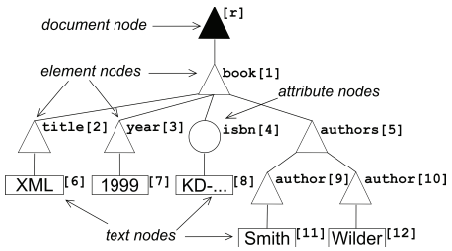

Figure 3. SXCCP+ data model

Stylesheet Language Transformations) and for query and modifications languages like XQuery (with update extensions) or XUpdate. In numerous information systems DOM API standard and XPath standard coexist like two alternatives to XML document access.

3. SXCCP+

SXCCP+ protocol, presented in this section, is based on locking mechanism and two phase locking protocol. The main idea of the SXCCP+ is concurrency control, which will be independent of (and not assigned to) a particular XML document access method. We assume that every access to XML document can be translated into a set of primitive and indivisible equivalent operations. Only these primitive operations are taken into account by the SXCCP+. For example, we assume that every DOM API function, and every XPath operation, performed on XML document, can be expressed in one, or in a sequence of primitive low-level operations. The fact that protocol is founded on primitive operations makes it autonomous from specific XML document access method. In the further part of this section we describe the set of primitive operations of the SXCCP+.

3.1. Data model

Data model used by the SXCCP+ for concurrency control is a simple DOM tree extension. Original DOM tree is extended by text nodes of attribute nodes. These additional nodes contain text value of attributes. In data model we distinguish following node types: document nodes, element nodes, attribute nodes and text nodes (which contain text value of elements and attributes). For example, data model for XML document from Fig. 2 is given in Fig. 3. Additional node types, like processing instructions and comments, are treated in the same way as text nodes, and are therefore omitted in this paper.

The aim of introducing the new text node for attributes is to increase degree of concurrency and unification of protocol rules, which take place in the attribute and text node access. In the data model all nodes are lockable entities, on which

Table 2. Conversion of DOM API methods to primitive operations

DOM API methods	Equivalent primitive operations
`n.firstChild`	`C(n); T(first(n))`
`n.previousSibling`	`T(previous(n))`
`n.getElementsByTagName`	`C(n); C(desc(n)); R(desc(n))`
`n.nodeName`	`R(n)`
`n.getAttribute(s)`	`C(n); R(attrs(n))`
`insertBefore(n',n)`	`I(n',parent(n),pos(n))`

concurrency control mechanism, due to execution of primitive operations, can acquire locks.

All primitive operations are performed transparently to users of DBMS by the concurrency control mechanism on the data model of XML document.

3.2. Primitive operations

We defined a set of primitive operations based on analysis of existing interfaces to XML document access. Before we introduce primitive operations, we define the concept of the node content. *Node content* is (according to node type) a tag (element node) or name of attribute (attribute node) or text value (text node) or node value (other node types which cannot have children nodes). In the SXCCP+ the following set of primitive operations is used: `C(n)` – test of child node existence, `T(n)` – node access (without access to its contents), `R(n)` – node content access, `U(n)` – node modification, it is an update of content of node, `D(n)` – node deletion, `I(n',n,l)` – insertion of `n'` node into `n` node at `l`th position of children of `n` node.

In order to use a SXCCP+ in a real XML database system, it is necessary to define the method of assigning the sequence of primitive operations equivalent to every operation existing in the used XML interfaces. In the case of procedural interfaces we can use association matrix, where we define sequence of equivalent primitive operations for each interface method. A fragment of such association matrix for DOM API is presented in Table 2. In the case of declarative interfaces, the sequence of equivalent primitive operations can be obtained according to the defined set of rules. We have defined such a set of rules for XPath-based interfaces. These rules are based on semantics of XPath expressions. Before we present these rules, we introduce some definitions partially based on definitions introduced in Jea, Chen and Wang (2002). As we mentioned in Section 2, each XPath location path L_j consists of location steps $S_{i,j}$, $1 \leq i \leq |L_j|$, where $|L_j|$ is the length of L_j.

The set of *context nodes* of a location step $S_{i,j}$ of location path L_j includes nodes that $S_{i,j}$ begins with. Context nodes are denoted by $C(S_{i,j})$. Context node of the first location step $S_{1,j}$ is a root of the document.

The set of *opening nodes* of a location step $S_{i,j}$ of location path L_j, denoted by $O(S_{i,j})$, includes nodes satisfying `axisname` in $S_{i,j}$.

The set of *mid-result nodes* of a location step $S_{i,j}$ of location path L_j, denoted by $M(S_{i,j})$, is the selection of $O(S_{i,j})$ satisfying `nodetest` in $S_{i,j}$.

The set of *result nodes* of a location step $S_{i,j}$ of location path L_j, denoted by $R(S_{i,j})$, is the selection of $O(S_{i,j})$ satisfying `predicate` in $S_{i,j}$. In fact, $R(S_{i,j}) = C(S_{i+1,j})$.

Let us notice that the predicate of a location step $S_{i,j}$ of location path L_j can be expressed by another location path $L_{i,j}$, which consists of other location steps $S_{k,i,j}$, $1 \leq k \leq |L_{i,j}|$. $C(S_{1,i,j})$ of location path $L_{i,j}$ is $M(S_{i,j})$.

The *destination nodes*, denoted by $N_d(L_j)$ of a location path L_j, are $R(S_{|Lj|,j})$.

As an illustration of these definitions let us analyze the following example. Assume that XPath location path has format
L_1=`/book[title/text()='XML']//author[1]`. Location path L_1 has two location steps $S_{1,1}$=`child::book[title/text()='XML']`, and
$S_{2,1}$=`descendant-or-self::author[1]`. Additionally, location step $S_{1,1}$ has predicate which uses location path $L_{1,1}$=`title/text()`. Location path $L_{1,1}$ has two location steps $S_{1,1,1}$=`child::title` and $S_{2,1,1}$=`child::text()`. Let us analyze *context*, *opening*, *mid-result* and *result* nodes of these location steps. Context node $C(S_{1,1})$ of location step $S_{1,1}$ is `[r]`. Opening node $O(S_{1,1})$ of the same location step is `[1]`. The same node (`[1]`) is an $M(S_{1,1})$. Before we get $R(S_{1,1})$ of a location step $S_{1,1}$, we have to evaluate its predicate. $C(S_{1,1,1})$ of location step $S_{1,1,1}$ is `[1]`. Opening nodes $O(S_{1,1,1})$ of the location step $S_{1,1,1}$ are `[2]`, `[3]`, `[5]`. $M(S_{1,1,1})$ and $R(S_{1,1,1})$ of location step $S_{1,1}$ is `[2]`. The same node is a $C(S_{2,1,1})$. Opening node $O(S_{2,1,1})$ of the location step $S_{2,1,1}$ is `[6]`. $M(S_{1,1,1})$ and $R(S_{1,1,1})$ of the same location step is also `[6]`. Value of `[6]` is `XML`, so predicate of $S_{1,1}$ is fulfilled and `[1]` is $R(S_{1,1})$ of the location step $S_{1,1}$. Node `[1]` is also the context node of the location step $S_{2,1}$. According to `axisname` format, the opening nodes $O(S_{2,1})$ of the location step $S_{2,1}$ are `[2]`, `[3]`, `[5]`, `[9]`, `[10]`, `[6]`, `[7]`, `[11]`, `[12]`. According to `testname` format of location step $S_{2,1}$, the mid-result nodes $M(S_{2,1})$ are `[9]` and `[10]`. Due to predicate of $S_{2,1}$ the result node $R(S_{2,1})$ is `[9]`. Location step $S_{2,1}$ is the last step of XPath location path L_1, therefore `[9]` node is also the destination node $N_d(L_1)$ of the location path L_1.

Now, we can present the set of rules, which assign the sequence of primitive operations equivalent to XPath location path L_j. In SXCCP+ protocol the sequence of primitive operations results from evaluation of every location step. At the beginning of the location step $S_{i,j}$ evaluation, operation `T` is performed on $C(S_{i,j})$. Then, if `axisname` has a `child`, `descendant` or `descendant-or-self` format[1], operation `C` is performed on $C(S_{i,j})$. Then, according to `nodetest` function, operation `T` or `R` is performed on $O(S_{i,j})$. If `nodetest` function is expressed by `*`, then `T` operation is performed, otherwise `R` operation is performed.

[1] Axisname which has `following` or `preceding` format as not allowed in SXCCP+ protocol

Table 3. Sequence of primitive operations equivalent to XPath-based update operations in SXCCP+ protocol

update operation	primitive operations	update operation	primitive operations
IB(n',n)	I(n',parent(n),pos(n)-1)	UT(n,t)	U(text(n))
IA(n',n)	I(n',parent(n),pos(n)+1)	RN(n',n)	D(n); I(n',parent(n),pos(n))
IF(n',n)	C(n); I(n',n,1)	DN(n)	D(n)
IL(n',n)	C(n); I(n',n,max(pos(childs(n)))	CN(n,t)	U(n)
IU(n',n,l)	C(n); I(n',n,l)		

```
(1) doc = getDocument();
(2) node = doc.getFirstChild();
(3) node = node.getLastChild();
(4) node = node.getLastChild();
(5) node.getNodeName();
(6) node = node.getFirstChild();
(7) node.setNodeValue('Speed');
```

Figure 4. DOM API example

```
for $i in /book[title/text()='XML']//author[1]
do rename $i as writer
```

Figure 5. XQuery example

As mentioned in Section 2, XPath expressions are used in many XPath-based interfaces which can read XML document fragments and also modify them. XQuery and XUpdate are one of the most popular. XQuery and XUpdate expressions allow modification of destination nodes $N_d(L_j)$, of location path L_j, by the following update operations: `IB(n',n)` – inserts new node `n'` before node `n`, `IA(n',n)` – inserts new node `n'` after node `n`, `IF(n',n)` – inserts new node `n'` as the first child of node `n`, `IL(n',n)` – inserts new node `n'` as the last child of node `n`, `IU(n',n,l)` – inserts new node `n'` as `l`th child of node `n`, `UT(n,t)` – updates text value of node `n` to `t` value, `RN(n',n)` – replaces node `n` to new node `n'`, `DN(n)` – deletes node `n`, `CN(n,t)` – changes node name `n` to `t` value. The sequence of primitive operations equivalent to each of update operations is presented in Table 3.

Let us analyze the following examples.

Exemplary DOM API operations are presented in Fig. 4. According to association matrix for DOM API, mentioned before, following primitive operations are equivalent to each DOM operation: (1): `T([r])`; (2): `C([r])`, `T(book[1])`; (3): `C(book[1])`, `T(authors[5])`; (4): `C(authors[5])`, `T(author[10])`; (5): `R(author[10])`; (6): `C(author[10])`, `T([12])`; (7): `U([12])`. An exemplary XQuery expression is presented in Fig. 5. XPath location path used in this expression has format L_1=`/book[title/text()='XML']//author[1]`. Its location steps have been described above. At the destination nodes of location

Table 4. Context, opening, mid-result, results nodes and corresponding primitive operations

S	$C(S)$	$O(S)$	$M(S)$	$R(S)$	primitive operations
$S_{1,1}$	`[r]`	`book[1]`	`book[1]`	`book[1]`	`T([r])`, `R([1])`
$S_{1,1,1}$	`book[1]`	`title[2]`, `year[3]`, `authors[5]`	`title[2]`	`title[2]`	`T([1])`, `R([2])`, `R([3])`, `R([5])`
$S_{2,1,1}$	`title[2]`	`[6]`	`[6]`	`[6]`	`T([2])`, `R([6])`
$S_{2,1}$	`book[1]`	`title[2]`, `year[3]`, `authors[5]`, `author[9]`, `author[10]`, `[6]`, `[7]`, `[11]`, `[12]`	`author[9]`, `author[10]`	`author[9]`	`T([1])`, `R([2])`, `R([3])`, `R([5])`, `R([9])`, `R([10])`, `R([6])`, `R([7])`, `R([11])`, `R([12])`
`CN`	$N_d(L_1)$: `author[9]`				`U([9])`

path L_1 XQuery expression changes node names – performs `CN` operations. Let us analyze primitive operations assigned to location step $S_{1,1}$. Operation `T` is performed on $C(S_{1,1})$ nodes – `[r]`. According to `axisname` format, `C` operation is performed on `[r]` node. The opening node $O(S_{1,1})$ of the location step $S_{1,1}$ is `[1]`, and according to `nodetest` format `R` operation is performed on this node. The same node (`[1]`) is an $M(S_{1,1})$. Before we get $R(S_{1,1})$, we have to evaluate its predicate. $C(S_{1,1,1})$ node of location step $S_{1,1,1}$ is `[1]`, and `T` operation and then `C` operation are performed on this node. The opening nodes $O(S_{1,1,1})$ of the location step $S_{1,1,1}$ are `[2]`, `[3]`, `[5]`, and `R` operation is performed (according to `textnode` format) on these nodes. $M(S_{1,1,1})$ and $R(S_{1,1,1})$ of location step $S_{1,1}$ is `[2]`. The same node is a $C(S_{2,1,1})$, then `T` and `C` (according to `axisname` format) operation is performed on this node. Opening node $O(S_{2,1,1})$ of location step $S_{2,1,1}$ is `[6]`, and `R` operation is performed on this node. Context, opening, mid-result, results nodes and corresponding primitive operation for analyzed XQuery expression are presented in Table 4.

3.3. Lock modes

Concurrency control mechanism of SXCCP+ is based on the two phase locking method. For all the primitive operations, introduced in preceding subsection, corresponding *operational basic lock mode* was defined. Thus, SXCCP+ uses the following operational basic lock modes: `LC` – child test lock, `LT` – traversal lock, `LR` – read lock, `LU` – update lock, `LD` – delete lock, `LW` – write lock, corresponding to primitive operations: `C`, `T`, `R`, `U`, `D`, `I`, respectively.

Additionally, we introduce the set of *operational tree lock modes*. These lock modes prevent lock escalation when performed operation has descendant nodes range. For example, in XQuery expression presented in Fig. 5, location step $S_{2,1}$=`descendant-or-self::author` implies `R` operation performed on all descendant nodes of `n` (`[1]`) node. In such cases, instead of acquiring operational basic locks on all descendant nodes of `n` node, SXCCP+ acquires operational tree lock on `n` node only. SXCCP+ uses the following operational tree lock modes: `LCC` – child test tree lock, `LTT` – traversal tree lock, `LRR` – read tree lock, `LUU` – update tree lock, `LDD` – delete tree lock, `LWW` – write tree lock.

Additionally, for each operational basic lock mode as well as for each operational tree lock mode corresponding *intentional lock mode* was defined. Therefore, we have the following intentional lock modes: `LIC` – intentional child test tree lock, `LIT` – intentional traversal lock, `LIR` – intentional read lock, `LIU` – intentional update lock, `LID` – intentional delete lock, `LIW` – intentional write lock; for operational basic lock modes: `LC`, `LT`, `LR`, `LU`, `LD`, `LW`, respectively, and `LICC` – intentional child test tree lock, `LITT` – intentional traversal tree lock, `LIRR` – intentional read tree lock, `LIUU` – intentional update tree lock, `LIDD` – intentional delete tree lock, `LIWW` – intentional write tree lock; for operational tree lock modes: `LCC`, `LTT`, `LRR`, `LUU`, `LDD`, `LWW`, respectively. Additionally, SXCCP+ uses two intentional lock modes `LICW` and `LICWW`. Meaning of these lock modes is presented in the next subsection.

3.4. Locking rules

Concurrency control mechanism controls every transaction, which performs access to documents in XML database system. SXCCP+ for every primitive operation (implied by operations of transactions) acquires locks with corresponding modes in two phases: the *phase of intentional locks* which is followed by the *phase of operational locks*. When primitive operation is performed on `n` node, during the phase of intentional locks, the intentional locks are acquired on all ancestor nodes of `n` node. The intentional locks are acquired from root of document to parent node order. When all required intentional locks are granted, operational lock is acquired on `n` node. When any of required locks can not be acquired due to incompatibility with other locks acquired by other transactions, then transaction and its locking mechanism stop, and wait for the release of incompatible lock. When locking mechanism stops, all previously acquired locks are held. Mode of the intentional locks, as well as mode of the operational locks, correspond to the type of primitive operation, which is performed on XML document, and its range. For example, the XQuery update expression of Fig. 5 implies `U` operation performed on the `author[9]` node. Thus, it requires intentional `LIU` locks on: `[r]`, `book[1]` and `authors[5]` node, then, it requires `LU` lock on `author[9]` node.

There is slightly different situation when `I` operation is performed. In this case, two intentional lock modes `LIW` and `LICW` (`LIWW` and `LICWW`, when operation

Table 5. Locks acquired by SXCCP+ protocol

primitive operation	range of operation	acquired locks in SXCCP+ protocol
C	n	LIC(ancestor(n)), LC(n)
C	desc(n)	LICC(ancestor(n)), LCC(n)
T	n	LIT(ancestor(n)), LT(n)
T	desc(n)	LITT(ancestor(n)), LTT(n)
R	n	LIR(ancestor(n)), LR(n)
R	desc(n)	LIRR(ancestor(n)), LRR(n)
U	n	LIU(ancestor(n)), LU(n)
U	desc(n)	LIUU(ancestor(n)), LUU(n)
D	n	LID(ancestor(n)), LD(n)
D	desc(n)	LIDD(ancestor(n)), LDD(n)
I	n	LIW(ancestor(n)-parent(n)), LICW(parent(n)), LW(n)
I	desc(n)	LIWW(ancestor(n)-parent(n)), LICWW(parent(n)), LWW(n)

has descendant nodes range) are used. During the phase of intentional locks LIW (LIWW) lock mode are acquired on all ancestor nodes except for parent node. Then LICW (LICWW) lock mode is acquired on a parent node. Lock modes LICW and LICWW are used to avoid phantom anomaly. Table 5 presents locks acquired by SXCCP+ protocol according to primitive operations and their range.

3.5. Lock matrix compatibility

As a result of the analysis of commutation of primitive operations, the complete compatibility matrix was built. It is presented in Table 6. As a result, the analysis of the complete lock compatibility matrix gives the following equivalence classes of lock modes: EQ_1= {LT, LIC, LIT, LICC, LITT}, EQ_2= {LTT, LCC}, EQ_3= {LIR, LIRR}, EQ_4= {LIU, LIUU}, EQ_5= {LIW, LID, LIWW, LIDD}, EQ_6= {LW, LD, LDD, LWW}, EQ_7= {LICW, LICWW}. Therefore, we introduce a representative for each equivalence class. Let the representative of equivalence classes EQ_1, EQ_2, EQ_3, EQ_4, EQ_5, EQ_6, EQ_7 be lock modes LT, LTT, LIR, LIU, LIW, LW and LICW, respectively. Lock modes not included in the presented equivalence classes are, in fact, members of the one-element equivalence classes, and they representatives of these classes. After this introduction we can present in Table 7 the lock compatibility matrix of SXCCP+ with the use of the representatives of equivalence classes.

Table 6. Complete compatibility matrix of SXCCP+

	granted																											
requested	LC	LT	LR	LU	LD	LW	LCC	LTT	LRR	LUU	LDD	LWW	LIC	LIT	LIR	LIU	LID	LICW	LIW	LICC	LITT	LIRR	LIUU	LIDD	LICWW	LIWW
LC	+	+	+	+	−	−	+	+	+	+	−	−	+	+	+	+	+	−	+	+	+	+	+	+	−	+
LT	+	+	+	+	−	−	+	+	+	+	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LR	+	+	+	−	−	−	+	+	+	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LU	+	+	−	−	−	−	+	+	−	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LD	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
LW	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
LCC	+	+	+	+	−	−	+	+	+	+	−	−	+	+	+	+	−	−	−	+	+	+	+	−	−	−
LTT	+	+	+	+	−	−	+	+	+	+	−	−	+	+	+	+	−	−	−	+	+	+	+	−	−	−
LRR	+	+	+	−	−	−	+	+	+	−	−	−	+	+	+	−	−	−	−	+	+	+	−	−	−	−
LUU	+	+	−	−	−	−	+	+	−	−	−	−	+	+	−	−	−	−	−	+	+	−	−	−	−	−
LDD	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
LWW	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
LIC	+	+	+	+	−	−	+	+	+	+	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LIT	+	+	+	+	−	−	+	+	+	+	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LIR	+	+	+	+	−	−	+	+	+	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LIU	+	+	+	+	−	−	+	+	−	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LID	+	+	+	+	−	−	−	−	−	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LICW	−	+	+	+	−	−	−	−	−	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LIW	+	+	+	+	−	−	−	−	−	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LICC	+	+	+	+	−	−	+	+	+	+	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LITT	+	+	+	+	−	−	+	+	+	+	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LIRR	+	+	+	+	−	−	+	+	+	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LIUU	+	+	+	+	−	−	+	+	−	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LIDD	+	+	+	+	−	−	−	−	−	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LICWW	−	+	+	+	−	−	−	−	−	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+
LIWW	+	+	+	+	−	−	−	−	−	−	−	−	+	+	+	+	+	+	+	+	+	+	+	+	+	+

Table 7. Lock compatibility matrix of SXCCP+

	granted											
requested	LT	LC	LR	LU	LW	LTT	LRR	LUU	LIR	LIU	LIW	LICW
LT	+	+	+	+	−	+	+	+	+	+	+	+
LC	+	+	+	+	−	+	+	+	+	+	+	−
LR	+	+	+	−	−	+	+	−	+	+	+	+
LU	+	+	−	−	−	+	−	−	+	+	+	+
LW	−	−	−	−	−	−	−	−	−	−	−	−
LTT	+	+	+	+	−	+	+	+	+	+	−	−
LRR	+	+	+	−	−	+	+	−	+	−	−	−
LUU	+	+	−	−	−	+	−	−	−	−	−	−
LIR	+	+	+	+	−	+	+	−	+	+	+	+
LIU	+	+	+	+	−	+	−	−	+	+	+	+
LIW	+	+	+	+	−	−	−	−	+	+	+	+
LICW	+	−	+	+	−	−	−	−	+	+	+	+

Lock modes, which are members of an equivalence class, are replaced in SXCCP+ by equivalence class representative. For example, `D` operation performed according to XQuery update expression `do delete /book/title`, requires `LIW` locks, instead of `LID` locks, on document node and `book[1]` node as well as `LW` lock, instead of `LD` lock, on `title[2]` node. This is due to the fact that `LID` lock mode is a member of EQ_5, where `LIW` lock mode is the equivalence class representative, and `LD` lock mode is a member of EQ_6 where `LW` lock mode is the equivalence class representative.

Additionally, SXCCP+ can use lock modes which are the combinational lock modes, and which could be used with the conversion of locks. Due to page number limitation these lock modes, lock conversion matrix and lock compatibility matrix of combinational lock modes are not presented in this paper.

Let us analyze the following example. Assume that transactions T_1 and T_2 are performed concurrently. Transaction T_1 executes DOM API operations presented in Fig. 4, whereas transaction T_2 executes XQuery expression presented in Fig. 5. Table 8 presents an example of their realization and required locks. Realization from 1 to 4 executes smoothly – only node access (`T`) and node content access (`R`) operations are performed. Their locks are compatible. In the fifth case, transaction T_1 performs `U` operation on `[12]` node, which requires `LIU` lock mode on `[r]`, `[1]`, `[5]`, `[10]` nodes. Unfortunately, transaction T_2 acquired `LRR` lock mode on `[4]` node. According to compatibility matrix (Table 7) `LRR` lock mode is incompatible with `LIU` lock mode. Therefore, transaction T_1 stops until transaction T_2 ends in 6.

Table 8. Concurrent realization of transactions T_1 and T_2

no.	operation	acquired locks	no.	operation	acquired locks
1	T_1: (1)–(3)	LT([r]), LC([r]), LIT([r]), LT([1]), LIC([r]), LC([1]), LIT([1]), LT([5])	4	T_2: $S_{2,1}$	LT([1]), LRR([1])
2	T_2: $S_{1,1}$, $S_{1,1,1}$, $S_{2,1,1}$	LT([r]), LIR([r]), LR([1]), LIT([r]), LT([1]), LIR([1]), LR([2]), LR([3]), LR([5]), LIT([1]), LT([2]), LIR([2]), R([6])	5	T_1: (6)–(7)	LIC([5]), LC([10]), LIT([10]), LT([12]), LIU([r],[1],[5], [10]), LU([12])
3	T_1: (4)–(5)	LIC([1]), LC([5]), LIT([5]), LT([10]), LIR([5]), R([10])	6	T_2: CN	LIU([r],[1],[5]), LU([9])

4. Correctness of SXCCP+

Before the formal proof we introduce some helpful definitions. The idea of the proof is based on Pleshachkov, Chardin and Kuznetsov (2005).

DEFINITION 1 *Let P be the set of primitive operations used by the SXCCP+ algorithm and defined in subsection 3.2.*

DEFINITION 2 *Let* Z *be a primitive operation, which completes the execution of a transaction. It is the last operation of each transaction and it releases all locks acquired by the transaction.*

DEFINITION 3 *Let op be an operation of transaction, which is one of the P operations, completed by* Z *operation – $op \in \{\mathtt{C}, \mathtt{T}, \mathtt{R}, \mathtt{U}, \mathtt{D}, \mathtt{I}\} \cup \{\mathtt{Z}\}$.*

DEFINITION 4 *Let action be a pair $a(op, t)$; where t is a transaction identifier.*

DEFINITION 5 *Let T be a transaction, which is a finite list of actions that have the same identifier of transaction.*

DEFINITION 6 *Let Q be a set of primitive operations, which do not change the document – $Q \in \{\mathtt{C}, \mathtt{T}, \mathtt{R}\}$.*

DEFINITION 7 *Let schedule S be a sequence of actions, which belong to the set of transactions.*

DEFINITION 8 *Let L_n^S be a set of locks obtained by transactions after processing the n-th step of S.*

DEFINITION 9 *Schedule S is legal if and only if at any step n of schedule S, a set of obtained locks L_n^S contains only compatible locks.*

DEFINITION 10 *Two schedules S and S' are equivalent if following conditions are fulfilled: (1) the schedule S is a permutation of S', preserving the order of actions in each transaction, (2) all the queries in the schedule S return the same results as the corresponding queries in S', (3) the resulting document in the schedule S and in the schedule S' is the same.*

DEFINITION 11 *Schedule S is serializable, if it is equivalent to any of the serial schedules S^{serial}.*

The schedule S generated by SXCCP+ is serialized according to the following partial order, transaction T_i precedes transaction T_j ($T_i < T_j$) when `Z` operation of T_i precedes `Z` operation of T_j or there is no `Z` operation of T_j in S.

Let S' be a schedule resulting from schedule S by swapping two subsequent actions, which belong to different transactions – $a(op_n, T_i)$ and $a(op_{n+1}, T_j)$ where $T_j < T_i$ in schedule S.

Let $l_n^S(op_n)$ be a set of locks acquired or released by operation op_n in S.

Let D_n^S be a resulting document after op_n ($n-th$ step of S).

Let $N_d(op_n)$ be destination nodes of op_n.

Let $ChildOf(N)$ be child nodes of N nodes, let $DescOf(N)$ be descendant nodes of N nodes and let $ParentOf(N)$ be parent nodes of N nodes.

LEMMA 1 *If S is a legal schedule then S' is also a legal schedule.*

Proof. S' is a legal schedule if the following conditions are satisfied: (1) all locks in $L_n^{S'}$ are compatible, (2) $L_{n+1}^{S'} \subseteq L_{n+1}^S$

- $op_n, op_{n+1} \in P$. Since operations which belong to P do not remove locks, we have $L_{n+1}^S = L_{n-1}^S \cup l_n^S(op_n) \cup l_{n+1}^S(op_{n+1})$ and $L_{n+1}^{S'} = L_{n-1}^{S'} \cup l_n^{S'}(op_{n+1}) \cup l_{n+1}^{S'}(op_n)$.
  If $op_n, op_{n+1} \in Q$. Operations do not change the document and locks acquired by them are compatible. It follows that $l_{n+1}^{S'}(op_n) = l_n^S(op_n)$ and $l_n^{S'}(op_{n+1}) = l_{n+1}^S(op_{n+1})$.
  If $op_n, op_{n+1} \in P$. The only case $l_{n+1}^{S'}(op_n) \neq l_n^S(op_n)$ ($l_n^{S'}(op_{n+1}) \neq l_{n+1}^S(op_{n+1})$) occurs when $N_d(op_n)$ ($N_d(op_{n+1})$) contains nodes inserted or deleted by op_{n+1} (op_n). This is the case when $op_n \in P, op_{n+1} \in \{\mathtt{D}, \mathtt{I}\}$($op_{n+1} \in P, op_n \in \{\mathtt{D}, \mathtt{I}\}$). Since all locks `LC`, `LT`, `LR`, `LU`, `LW` are not compatible with `LW`, and all locks `LTT`, `LRR`, `LUU`, `LW` (operations with descending range) are not compatible with `LIW` and `LW`, and each of locks `LT`, `LIR`, `LIU`, `LIW`, `LICW` is not compatible `LW` (operations with descending range) we have contradiction.
  We have $L_n^{S'} = L_n^S$ and $L_{n+1}^{S'} = L_{n+1}^S$. Thus, we proved (1) and (2).

- $op_{n+1} \in \texttt{Z}$. Since Z removes locks, it follows that $L^S_{n+1} = L^S_{n-1} \cup l^S_n(op_n) \setminus l^S_{n+1}(op_{n+1})$ and $L^{S'}_n = L^{S'}_{n-1} \setminus l^{S'}_n(op_{n+1})$. Due to $l^S_{n+1}(op_{n+1}) = l^{S'}_n(op_{n+1})$ we have $L^{S'}_n \subset L^S_{n+1}$. Thus we proved (1).
  The only case when $L^{S'}_{n+1} \neq L^S_{n+1}$ is when Z terminates transaction, which removes $N_d(op_n)$. This is the case when S is not a legal schedule because none of locks LC, LT, LR, LU, LW (LTT, LRR, LUU, LW; LT, LIR, LIU, LIW, LICW) is compatible with LW (LIW, LW; LW). Thus, we proved also (2). ∎

LEMMA 2 *If S is a legal schedule and $op_n \in Q$ or $op_{n+1} \in Q$, then their results in S' are the same.*

Proof. Due to the fact that Q operations do not change the document, we have to consider the combinations of Q operations and $P \setminus Q$ operations.

- $(\{\texttt{I},\texttt{D}\},\texttt{T}) \rightarrow (\texttt{T},\{\texttt{I},\texttt{D}\})$ $((\{\texttt{I},\texttt{D}\},\texttt{R}) \rightarrow (\texttt{R},\{\texttt{I},\texttt{D}\}))$. I or D changes the results of T(R) in three cases:
  - if $N_d(op_n) \cap N_d(op_{n+1}) \neq \emptyset$ then LW and LT(LR) are obtained by different transaction on the same node. We have contradiction,
  - $N_d(op_{n+1}) \subseteq DescOf(N_d(op_n))$. It is an impossible case, because LT(LIR) and LW are incompatible,
  - $N_d(op_n) \subseteq DescOf(N_d(op_{n+1}))$ and op_{n+1} has descendent range. It is an impossible case, because of incompatibility of LIW and LTT(LRR).
- $(\texttt{T},\texttt{D}) \rightarrow (\texttt{D},\texttt{T})$ $((\texttt{R},\texttt{D}) \rightarrow (\texttt{D},\texttt{R}))$. See previous point.
- $(\texttt{T},\texttt{I}) \rightarrow (\texttt{I},\texttt{T})$ $((\texttt{R},\texttt{I}) \rightarrow (\texttt{I},\texttt{R});\ (\texttt{C},\texttt{I}) \rightarrow (\texttt{I},\texttt{C}))$. Operation I can change results of T (R; C) when I inserts nodes, which belong to destination nodes of T (R; C) in S'. In SXCCP+ every access to child nodes of n node requires LC lock on n node (C operation must be performed on n node). We have to consider two cases:
  - $N_d(op_{n+1})$ are child nodes of node on which C (op_n) operation was performed. Due to incompatibility LC and LICW this case is impossible,
  - $N_d(op_{n+1}) \subseteq DescOf(N_d(op_n))$ and op_n has descendent range. LTT (LRR; LTT) and LIW are not compatible – thus, it is an impossible case.
- $(\{\texttt{I},\texttt{D}\},\texttt{C}) \rightarrow (\texttt{C},\{\texttt{I},\texttt{D}\})$. I or D change the results of C in the following cases:
  - $N_d(op_n) \cap N_d(op_{n+1}) \neq \emptyset$. LW and LC are not compatible, therefore this case is impossible,
  - $N_d(op_{n+1}) \subseteq DescOf(N_d(op_n))$. It is an impossible case, because LT(LIR) and LW are incompatible,

  - $N_d(op_n) \subseteq ChildOf(N_d(op_{n+1}))$. Due to incompatibility of LC and LICW this case is impossible.
  - $N_d(op_n) \subseteq DescOf(N_d(op_{n+1}))$ and op_{n+1} has descendent range. It is an impossible case, because of incompatibility of LIW and LTT.

- $(\mathtt{C}, \mathtt{D}) \rightarrow (\mathtt{D}, \mathtt{C})$. See previous point.
- $(\mathtt{U}, \{\mathtt{C}, \mathtt{T}\}) \rightarrow (\mathtt{U}, \{\mathtt{C}, \mathtt{T}\})$. U operation does not change the number of nodes in the document; therefore it can not change the outcome of the C and T operations.
- $(\{\mathtt{C}, \mathtt{T}\}, \mathtt{U}) \rightarrow (\{\mathtt{C}, \mathtt{T}\}, \mathtt{U})$. See previous point.
- $(\mathtt{U}, \mathtt{R}) \rightarrow (\mathtt{R}, \mathtt{U})$. U operation can change the result of R operation in the following cases:
  - if $N_d(op_n) \cap N_d(op_{n+1}) \neq \emptyset$, then LU lock and LR lock are obtained by different transaction on the same node. We have contradiction,
  - $N_d(op_{n+1}) \subseteq DescOf(N_d(op_n))$ and op_n has descendent range. Impossible case due to LIR and LUU incompatibility,
  - $N_d(op_n) \subseteq DescOf(N_d(op_{n+1}))$ and op_{n+1} has descendent range. LIU and LRR are not compatible, therefore this case is impossible.
- $(\mathtt{R}, \mathtt{U}) \rightarrow (\mathtt{U}, \mathtt{R})$. See previous point. ∎

LEMMA 3 *If S is a legal schedule, then $D^S_{n+1} = D^{S'}_{n+1}$.*

Proof. We have to consider combinations of $P \setminus Q$ operations. Operations Q do not change the state of the document, therefore, when op_n or op_{n+1} belongs to Q, then D^S_{n+1} and $D^{S'}_{n+1}$are the same.

- $(\mathtt{U}, \mathtt{U}) \rightarrow (\mathtt{U}, \mathtt{U})$. U operations do not commute in the following cases:
  - if $N_d(op_n) \cap N_d(op_{n+1}) \neq \emptyset$, then LU locks are obtained by different transaction on the same node. We have contradiction,
  - $N_d(op_{n+1}) \subseteq DescOf(N_d(op_n))$ and op_n has descendent range. It is an impossible case due to LUU and LIU incompatibility,
  - $N_d(op_n) \subseteq DescOf(N_d(op_{n+1}))$ and op_{n+1} has descendent range. It is an impossible case due to LIU and LUU incompatibility.
- $(\mathtt{U}, \mathtt{D}) \rightarrow (\mathtt{D}, \mathtt{U})$ $((\mathtt{U}, \mathtt{I}) \rightarrow (\mathtt{I}, \mathtt{U}))$. U and D(I) operations do not commute in the following cases:
  - if $N_d(op_n) \cap N_d(op_{n+1}) \neq \emptyset$, then LU lock and LW lock are obtained by different transactions on the same node. We have contradiction,
  - $N_d(op_n) \subseteq DescOf(N_d(op_{n+1}))$ and op_{n+1} has descendent range. It is an impossible case due to LW and LUU incompatibility,
  - $N_d(op_{n+1}) \subseteq DescOf(N_d(op_n))$ and op_n has descendent range. It is an impossible case due to LUU and LIW incompatibility.

- $(\texttt{D},\texttt{U}) \rightarrow (\texttt{U},\texttt{D})$ ($(\texttt{I},\texttt{U}) \rightarrow (\texttt{U},\texttt{I})$). Similar to previous point.
- $(\texttt{D},\texttt{D}) \rightarrow (\texttt{D},\texttt{D})$. These operations commute.
- $(\texttt{D},\texttt{I}) \rightarrow (\texttt{I},\texttt{D})$. Operations D and I do not commute in the following cases:
  - if $N_d(op_n) \cap N_d(op_{n+1}) \neq \emptyset$, then LW locks are obtained by different transaction on the same node. We have contradiction,
  - $N_d(op_{n+1}) \subseteq DescOf(N_d(op_n))$ and op_n has descendent range. It is an impossible case due to LIW and LW incompatibility,
  - $N_d(op_n) \subseteq DescOf(N_d(op_{n+1}))$ and op_{n+1} has descendent range. LW and LIW are not compatible, therefore this case is impossible.
- $(\texttt{W},\texttt{D}) \rightarrow (\texttt{D},\texttt{W})$. Similar to previous point.
- $(\texttt{I},\texttt{I}) \rightarrow (\texttt{I},\texttt{I})$. I operations do not commute in the following cases:
  - if $ParentOf(N_d(op_n)) \cap ParentOf(N_d(op_{n+1})) \neq \emptyset$, then LC and LICW locks are obtained by different transactions on the same node. We have contradiction,
  - $N_d(op_n) \subseteq DescOf(N_d(op_{n+1}))$ and op_{n+1} has descendent range. It is an impossible case due to LW and LIW incompatibility,
  - $N_d(op_{n+1}) \subseteq DescOf(N_d(op_n))$ and op_n has descendent range. LIW and LW are not compatible, therefore this case is impossible.
- $(\{\texttt{U},\texttt{D},\texttt{I}\},\texttt{Z}) \rightarrow (\texttt{Z},\{\texttt{U},\texttt{D},\texttt{I}\})$ – These operations commute.

We have considered all possible combinations of operations op_n and op_{n+1}. In all of these combinations $D^S_{n+1} = D^{S'}_{n+1}$. ∎

THEOREM 1 *All schedules S generated by SXCCP+ are serializable.*

Proof. Every legal schedule S can be reduced to serial schedule S^{serial}. It is possible through multiple swapping of two subsequent actions $a(op_n, T_i)$ and $a(op_{n+1}, T_j)$ where $T_j < T_i$. Schedule S is serial schedule S^{serial} if there are no such pairs of actions.

Taking into account Lemmas 2 and 3, we obtain that schedules S and S' are equivalent. In Lemma 1 we proved that S' is a legal schedule. Schedule S' can be used to obtain $(S')'$. Taking into account the fact that schedule S is a finite list of actions, we conclude that schedule S can be reduced to S^{serial} by finite number of swaps of actions. ∎

5. Experimental results

Our experiments were conducted with the support of XML database system prototype, which was created at the Institute of Computing Science of the Poznan University of Technology. This prototype was constructed for the purpose of research on concurrency control mechanisms in XML database systems (Siekierski and Siekierska-Wojnowska, 2005).

5.1. Tested protocols

We conducted our experiments for the following concurrency control protocols: NO2PL, Node2PL, OO2PL, taDOM and SXCCP+. Protocols NO2PL, Node2PL, OO2PL, introduced in Helmer (2004) have two versions: basic and extended. In extended versions of these protocols additional locks were introduced. These additional locks correspond to the read content node and modification node operations. We performed experiments with extended versions of these protocols. Protocol taDOM has a special lock mode U, which supports read with potential write access and prevents granting further read locks. This lock mode decreases deadlock probability, but also decreases the degree of concurrency. Similar lock modes do not exist with other tested protocols, although they could. Therefore, in our tests we use the version of taDOM protocol without the U lock mode. Additionally, due to lack of lock conversion tables in NO2PL, Node2PL and OO2PL protocols, we do not do such conversions. This fact has an influence on the number of acquired locks, but in this way this number is comparable among different protocols.

5.2. Characteristics of the XML document

Our experiments were performed on several types of XML documents. We present results of experiments performed on one XML document bench0065.xml, which was generated by xmlgen – The Benchmark Data Generator, created as a part of XMark – An XML Benchmark Project (Schmidt, 2002).

5.3. Transaction classes

Access to XML documents was realized by the set of transactions, which were concurrently started. Cardinality of the transactions set was changed from 1 to 49. Operations performed by transactions were based on DOM API. Each transaction was one out of four transaction classes. The choice of transaction class for each transaction was random with the same probability. Each of transaction classes has characteristics as follows:

- OneDocumentPointModify – transaction navigates from root node to one of leaf nodes, and modifies its content. Traverse from one level of XML document to a deeper one is realized by the choice of direction (GetFirstChild or GetLastChild), and then by the move through random number of sibling nodes. At the last node of each traverse, transaction performs node content access (reads its tag). At the end, transaction performs node content modification, which depends on node type.
- OneDocumentRandomLevelModify – transaction navigates from root node to node on one of random levels, and modifies the node content. Traverse from one level of XML document to a deeper one is realized in the same way as in OneDocumentPointModify transaction class. For each document

level, transaction can stop navigation with probability depending on the average depth of the document.
- OneDocumentRandomLevelDelete – transaction navigates from root node to node on one of random levels, and removes destination node. Navigation is realized in the same way as in OneDocumentRandomLevelModify class.
- OneDocumentRandomLevelInsert – transaction navigates from root node to node on one of random levels, and inserts a new node as a child or neighbor of destination node. Navigation is realized in the same way as in OneDocumentRandomLevelModify class.

5.4. Results

Presented figures reflect the results of performed experiments as the average values of the following chosen measures: test times (Fig. 6) – the time required to service all concurrent transactions as a function of cardinality of transaction set, number of conflicts (Fig. 7) – the number of conflict situations as a function of cardinality of transaction set. Each conflict situation was resolved by restart or waiting of transaction because of the fact that WAIT-DIE algorithm was used, number of locks held (Fig. 8) – the maximum number of locks held by the transactions as a function of cardinality of transaction set.

5.5. Analysis of the results

Let us take a closer look at the results. As we can see in Fig. 8, number of locks held by the OO2PL protocol is higher than in other protocols. This difference is due to the number of node locks, which are acquired for every operational lock in OO2PL. The number of locks held by SXCCP+ is not lower than in other protocols. It is because of the intentional locks. These intentional locks have crucial meaning because they enable SXCCP+ usage with other XML interfaces and they enable operational tree lock mode usage, which can decrease the number of locks in many cases (tree lock modes were not used in presented experiments). Figs. 6 and 7 reflect the fact that the SXCCP+ has medium ability to lead transaction to the successful realization, but SXCCP+ does not allow for phantom anomaly, and gives serializable realizations, and therefore it must be more restrictive than OO2PL, NO2PL or Node2PL, which allow phantom anomaly to occur.

6. Conclusions and future work

In this paper we have presented a new locking protocol for concurrency control access in XML database systems, named SXCCP+. It is the first protocol, which is not assigned to any particular XML interface. It is based on primitive and indivisible operations, which may be treated as components of any operations of any XML access interface. It means that SXCCP+ is the protocol, which

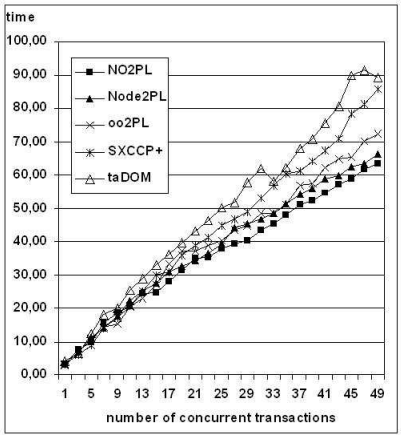

Figure 6. Test times

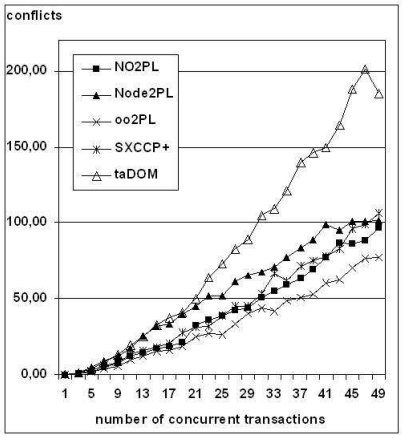

Figure 7. Number of conflicts

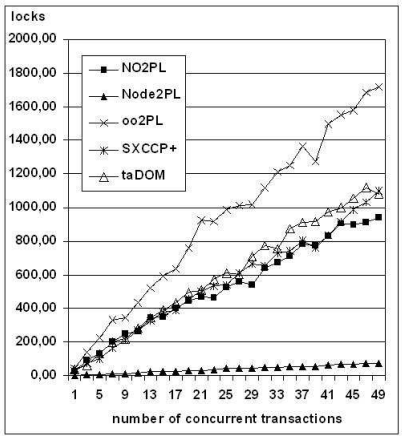

Figure 8. Number of locks held

can be treated as general and neither assigned to nor dependent of a particular XML interface. This fact has the key meaning for most XML database systems, when different interfaces coexist. Moreover, presented results of conducted experiments show that SXCCP+ is not worse than specialized protocols, assigned to particular interface, like taDOM, OO2PL, NO2PL, and Node2PL.

Our experiments presented in this work are focused on DOM API based protocols. Therefore, in our next work we will conduct series of experiments which compare SXCCP+ with other concurrency control protocols, which are based on XPath expressions. Then, we plan to introduce some modifications into SXCCP+ which, as results, will give higher degree of concurrency.

References

Bernstein, P.A., Hadzilacos, V. and Goodman, N. (1987) *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman, Boston, MA, USA.

Cellary, W., Gelenbe, E. and Morzy, T. (1988) *Concurrency Control in Distributed Database Systems.* Elsevier Science, New York, NY, USA.

Choi, E.H. and Kanai, T. (2003) XPath-based concurrency control for XML data. In: *Proceedings of the 14th Data Engineering Workshop (DEWS 2003)*, Kaga City, Ishikawa, Japan. IEICE website; www.ieice.org, 302–313.

Clark, J. and DeRose, S. (1999) XML path language (XPath) version 1.0. Technical report, World Wide Web Consortium. www.w3.org/TR/xpath

Dekeyser, S. and Hidders, J. (2002) Path locks for xml document collaboration. In: *WISE '02: Proceedings of the 3rd International Conference on Web Information Systems Engineering*, Washington, DC, USA. IEEE Computer Society, 105–114.

Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. (1976) The notions of consistency and predicate locks in a database system. *Commun. ACM* **19** (11), 624–633.

Grabs, T., Böhm, K. and Schek, H.J. (2002) Xmltm: Efficient transaction management for xml documents. In: *CIKM '02: Proceedings of the 11th International Conference on Information and Knowledge Management*, McLean, Virginia. ACM Press, New York, 142–152.

Haustein, M.P. and Härder, T. (2003) taDOM: A tailored synchronization concept with tunable lock granularity for the DOM API. In: *ADBIS 2003: Proc. of Advances in Databases and Information Systems.* **LNCS 2798**, Springer, 88–102.

Helmer, S., Kanne, C.C. andd Moerkotte, G. (2004) Evaluating lock-based protocols for cooperation on XML documents. *SIGMOD Record* **33** (1), 58–63.

Jankiewicz, K. (2006) Survey and analysis of concurrency control methods for xml database systems. In: *Proc. of the 5th Intern. Conference MISSI'06 - Multimedia and Network Information Systems*, Wroclaw, Poland. Wroclaw University of Technology Press, 51–66.

Jea, K.F., Chen, S.Y. and Wang, S.H. (2002) Concurrency control in XML document databases: XPath locking protocol. In: *ICPADS '02: Proceedings of the 9th Intern. Conference on Parallel and Distributed Systems.* IEEE Computer Society, Los Alamitos, CA, USA, 551-556.

Pleshachkov, P., Chardin, P. and Kuznetsov, S.D. (2005) A DataGuide-Based Concurrency Control Protocol for Cooperation on XML Data. In: *ADBIS 2005: Proc. of Advances in Databases and Information Systems.* **LNCS 3631**, Springer, 268–282.

Siekierski, M. and Siekierska-Wojnowska, A. (2005) Concurrency control for semistructural data. Master Thesis, Institute of Computer Science, Poznan University of Technology, Poznan.

Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I. and Busse, R. (2002) XMark: A benchmark for XML data management. In: *VLDB'02: Proc. of the 28th International Conference on Very Large Data Bases*, Hong Kong, China. VLDB Endowment, 974–985.