Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

XML Transactions

by

Pavel Strnad

A thesis submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

PhD programme: Electrical Engineering and Information Technology
Specialization: Computer Science and Engineering

May 2013

Thesis Supervisor:
Karel Richta
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13
121 35 Praha 2
Czech Republic

Thesis Co-Supervisor:
Michal Valenta
Department of Software Engineering
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Abstract and contributions

The field of XML and other related technologies has emerged a lot of research in past few years and the technologies based on XML has became industry standard nowadays. Therefore one of the most important areas of the research interest is the field of native XML databases. If we want to use a native XML database as a database with updates we need an XML update language, but in the area of XML update languages the situation was a long time unclear. Hopefully, despite of many existing proposals, the common update language, XQuery Update Facility (XQUF), come from the World Wide Web Consortium (W3C) and becomes a standard.

In this thesis, we focus primarily on the transaction processing in native XML databases. First, we introduce formal specification of transactions in XML and consequently we provide a basic description of XQUF. We show that it is possible to express formally the locking semantics of XQUF in terms of transactions. We propose an extension of XQUF that provides transaction specific features. We give a benchmark specification to measure performance of transaction processing in native XML databases (NXD).

The main contributions of the thesis are the following:

1. Formal specification of XML transactions is given.
2. XQUF semantics is described in terms of transactions.
3. XQUF is extended by transaction specific features.
4. Benchmark specification to measure performance in NXDs is described.

In conjunction with these theoretical outcomes we have also developed many working prototypes that were used as proof-of-concept implementations for our benchmarking experiments.

Keywords:

XML, transaction processing, transaction semantics, XQuery Update Facility semantics

Acknowledgements

First of all, I would like to express my gratitude to my thesis supervisor, Karel Richta. He has been a constant source of encouragement and insight during my research. He, together with Michal Valenta, provided me with numerous opportunities for professional advancements. His continued support is gratefully acknowledged. His efforts as thesis supervisor contributed substantially to the quality and completeness of the thesis. I have learned a great deal from them. Many other people influenced my work. I wish to thank to Pavel Loupal, Jan Vraný and Ondřej Macek.

The staff of our department has provided me a pleasant and flexible environment for my research. Especially, I would like to thank to doc. Šnorek – the head of the department – for taking care of my financial support. My work has been partially supported by grants from FRVS and GACR grant agencies.

Finally, my greatest thanks to my family and friends whose support was of great important during finishing the thesis.

Dedication

To my wife Petra

Contents

List of Figures

List of Tables

1 Introduction

XML language [7] designed by the consortium W3C [63] is currently the standard for exchanging and storing data. Its suitability for many applications lies in the fact that it is easily readable by the user and the computer. Its advantages include the ability to specify the domain and structure of stored data using the schema. XML is the world-wide language suitable to use anywhere where we need to separate the data from their presentation. With the growing number of XML documents the need for their efficient storage is increasing for the needs of searching information stored in them. As a convenient way to save documents is widely accepted using a database managemenent system (”database”). Bearing in mind that for some applications is not sufficient to store only documents, but also allow their effective update, ie. insert and delete stored information. It leads to build XML database where documents will be stored and indexed as in relational databases and their changes will be simple to realize. The difference between relational and XML data model is in the structure of their storage. The relational data model [16] stores data into interconnected ”tables”. In contrast, XML data model is organized in a tree structure. Relational databases are currently the widely used and accepted platform for storing large amounts of data. They are developed on the very strong theoretic background and are used since the second half of the seventies of the twentieth century. Most of the techniques developed and successfully used in relational databases can also be applied to XML databases. One of the important feature is a transaction processing of the stored data. Informally, we can say that we need to ensure that multiple users can simultaneously access the stored data. These users can not only read the data, but they also may change them while maintaining their consistency and availability.

This work deals with the transaction processing in the (native) XML databases from several perspectives. First two chapters of the thesis define the formal background of transaction processing and XML transactions respectively. The formal semantics of transaction processing in XML databases is given. This transaction semantics extends semantics of XQuery and XQuery Update Facility languages, which were created by the W3C to query and modify data stored in XML documents or databases. The original language specification of XQuery Update Facility does not include transaction processing. To verify the accuracy of the newly defined semantics we implemented its most important constructs in the Maude system, which is a formal verification tool for formal semantics. We extended the syntax and the semantics of XQuery Update Facility to cover the needs of transaction processing. We have done it by adding syntax constructs for transaction control into the language. The second part of the thesis deals with the question of benchmarking (native) XML database systems and measurement of their performance with respect to transaction processing. We designed a special benchmark for measuring the overhead of a transaction processing module.

During writing this work several different prototypes were implemented. We used them to verify results of our work. In particular, the prototype of the native XML database called CellStore was implemented in the Smalltalk programming language and is available for a free download [68]. The second prototype of a native XML database called RedXML [37] is implemented in Ruby and presents a proof-of-concept of storing XML documents into a key-value database Redis [51]. On this prototype were tested and evaluated various techniques of mapping XML documents into a key-value database. A large part of those prototypes was implemented by the bachelor and master students of the Department of Computer Science and Engineering.

1.1 Contributions

This thesis provides the following contributions:

1. We introduce detailed formal semantics of the XQuery Update Facility languge, a functional language standardized by W3C, extended by transaction processing. This semantics can be used for verification of concurrent programs using this language, moreover the semantics is suitable to be the part of the standard in the future.

2. We extend XQuery Update Facility syntax and semantics by expressions for transaction control. These expressions are needed to control program flow according to transaction processing. This extension is suitable to be the part of the standard in the future.

3. We provide a new simple benchmark for measuring overhead of a transaction manager module. This benchmark can be used for component based systems in the future.

4. We specify a transaction processing for XML-λ Language by mapping its operations into DOM operations and utilizing taDOM locking protocol.

5. We provide semantics verification by the prototype implementation in the Maude system. This prototype implementation is very useful for formal proving of algebraic features of the language such as confluence or coherence.

6. We provide a prototype implementation of the native XML database CellStore, which is used as a testbed for experiments.

1.2 Organization of the thesis

The thesis is divided into seven chapters. There are five main parts. The first part contains Chapter 1 which provides basic information about the thesis including the summary of contributions, the motivation and the organization of the thesis. The second part is divided into two chapters (Chapter 2, Chapter 3) and introduces theoretical background of transaction processing in relational and XML databases. The third part (Chapter 4) presents the transaction processing extension of the XQuery Update Facility semantics. The fourth part (Chapter 5) proposes a new type of benchmark which evaluates the performance of a (native) XML database system. The fifth part (Chapter 6) describes prototypes implemented during writing the thesis. Finally, in Appendices, we supply some additional materials related to the thesis, particularly complete syntax of the XQuery Update Facility extended by the transaction control language.

- *Chapter 1* contains a basic introduction and problem specification.

- *Chapter 2* introduces theoretical backgorund of transaction processing in databases.

- *Chapter 3* describes specific differences of transaction processing in XML databases and provides a related work in this area.

- *Chapter 4* introduces syntax and semantics of XQuery Update Facility. The new transaction semantics is presented in this chapter. The verification tool implemented in Maude is presented in this chapter.

- *Chapter 5* describes a new component benchmark specification targeted on transaction manager module overhead.

- *Chapter 6* contains description of implemented prototypes during writing this thesis.

- *Chapter 7* summarizes the thesis contributions and provides suggestions for the future research.

1.3 Conventions and notations

In the thesis we use the following notation:

$$\begin{aligned}
&\textbf{var}\ g : \mathcal{G}_{cont} \\
&\textbf{var}\ l : \mathcal{L}_{cont} \\
CC(g[CCL_{/l[TRANS]} &==< op : REST >], l) = \\
&= CC(OP(op, g[CCL_{/l[TRANS]} := REST], l))
\end{aligned}$$

This semantics equation defines function CC with two parameters g and l, where the type of g is $\mathcal{G}_{cont}$ and the type of l is $\mathcal{L}_{cont}$. We use *pattern matching* to match the left side of the equation, so if the pattern of the left side is satisfied then the equation can be applied to the expression. The pattern for variable g says:

$$g[CCL_{/l[TRANS]} ==< op : REST >]$$

It means that g contains a *CCL* filtered by the transaction (the transaction is stored in local context l in "variable" $TRANS$). *CCL* has to have the structure of the list $< op : REST >$, which has more than one item. The right side of the equation rewrites the left side of the equation to:

$$CC(OP(op, g[CCL_{/l[TRANS]} := REST], l))$$

The function CC is invoked with the OP function as the parameter. The OP function has tree parameters *op, g and l*, where g's "variable" *CCL* is modified, item *op* is reduced and *CCL* is set to *REST*. Indeed, this notation can be mapped to Semantics Definition in Section 4.2. The pattern $g[CCL ==< op : REST >]$ is equivalent to expression $getCCL(g) ==< op : REST >$, where the function $getCCL$ extracts CCL from g and $g[CCL := REST]$ can be translated to $setCCL(g, REST)$.

2 Background

This chapter provides the basic survey of concurrency control mechanisms and concepts used in (not only) relational databases. We put emphasis on basic principles of standard transaction theory, which utilizes lock primitives to ensure *correct* transaction execution. The major part of this chapter is based on definitions from well-known books [50, 23, 4].

2.1 Transaction Processing and Isolation Concepts

2.1.1 Overview

This section introduces the isolation definitions and theorems. The theorems state that transactions can execute in parallel with complete isolation if the objects of each transaction accesses and modifies are disjoint from those modified by others [23, 50]. We present the theorems which indicate how locking can achieve it. Refinements of these results can increase concurrency among transactions. The strategy presented in this section called granular locks or predicate locks [23] allows transactions to lock subsets of an object.

2.1.2 Transactions

2.1.2.1 Definitions

The following definitions of terms are adopted from [23] and [50]. These definitions have also general applicability for transactions in XML databases.

Definition 2.1.1. *A transaction is a sequence of actions starting with a BEGIN action followed by any combination of:*

- *READ(o)*
- *WRITE(o,value)*
- *XLOCK(o)*
- *SLOCK(o)*
- *UNLOCK(o)*

where o *is an object stored in the system and* value *is a corresponding value. A transaction ends with a COMMIT or ROLLBACK action. Transactions are represented by a sequence in the form* $<< t_i, a_i, o_i > | i = 1, .., n >$; *this means that the* ith *step of transaction* t *performed action* a_i *on object* o_i.

A *simple transaction* is composed of READ, WRITE, XLOCK, SLOCK, and UNLOCK actions [23]. Every transaction T can be translated into an *equivalent simple transaction* as follows [23]:

1. Discard the BEGIN action.
2. If the transaction ends with a COMMIT action, replace the action with the following sequence of UNLOCKs:

   $$< UNLOCK\ A | if\ SLOCK\ A\ or\ XLOCK\ appears\ in\ T\ for\ any\ object\ A >$$

   .
3. If the transaction ends with a ROLLBACK action, replace the action with the following sequence of WRITEs and UNLOCKs:

   $$< WRITE\ A | if\ WRITE\ A\ appears\ in\ T\ for\ any\ object\ A >$$

   $$< UNLOCK\ A | if\ SLOCK\ A\ or\ XLOCK\ appears\ in\ T\ for\ any\ object\ A >$$

   .

2.1.2.2 Flat Transactions

There exists many types of transactions. The most strict transactions are called *flat transactions* (ACID). These transactions have to fulfill ACID properties [23]:

- **Atomicity** – Transaction's changes to the state are atomic: either all happen or non happen. These changes include database changes, messages, and actions on transducers.
- **Consistency** – Transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program.

- **Isolation** – Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.
- **Durability** – Once a transaction completes successfully (commits), its changes to the state survive failures.

2.1.3 Transaction Dependencies

2.1.3.1 The Dependency Model of Isolation

Two READ actions by two different transactions to the same object cannot violate consistency because reads do not change the object state. Hence, it is only UPDATE and INSERT actions that may cause the problem. Two UPDATE actions to an object by the same transaction do not violate consistency because the ACID property assumes that the transaction knows what it is doing to its data. Consequently, only UPDATE or INSERT related interactions between two concurrent transactions can create inconsistency or violate isolation.

This fact can be expressed by letting I_i be the set of objects read by transaction T_i (its inputs), and O_i be the set of objects written by T_i (its outputs). The set of transactions T_i can run in parallel with no concurrency anomalies if their outputs are disjoint from one another's inputs and outputs [23]:

$$\forall i \neq j \; O_i \cap (I_i \cup O_j) = \emptyset$$

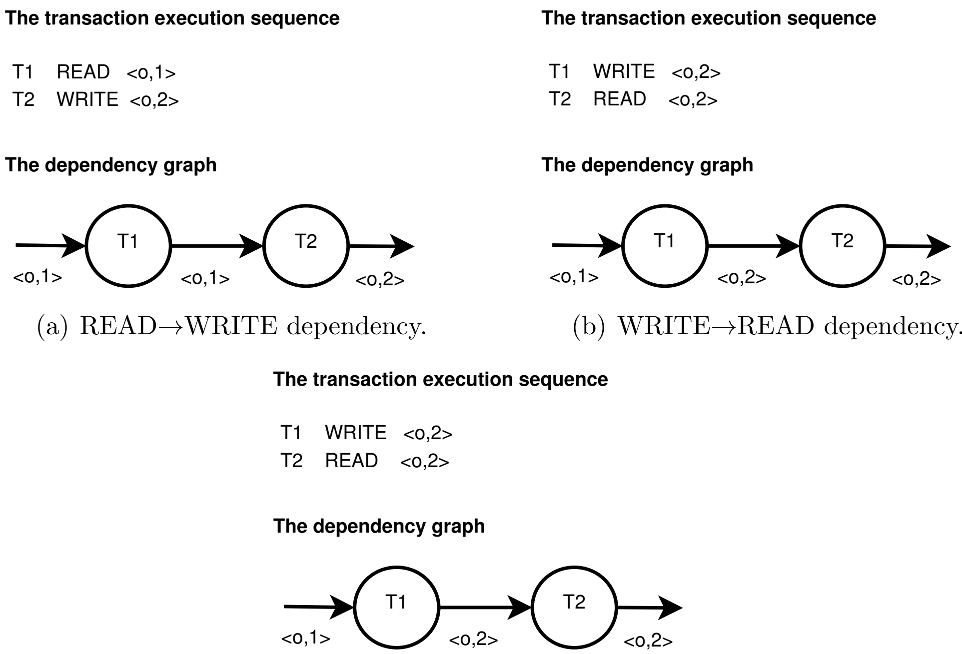
2.1.3.2 Transaction Dependencies

We assume the dynamic allocation model [23] that considers allocation of resources (objects) during the transaction. This means that the resource is allocated when it is needed. The older static allocation model expected allocation of resources before the transaction. This approach leads to execution of only one transaction at a time. The dynamic allocation model postulates that transactions are sequences of actions operating on objects.

Objects go through a sequence of *versions* as they are written by these actions. Reads do not change the object version, but each time the object is changed, it gets a new version. If a transaction reads an object, the transaction *depends* on that object version. If the transaction writes an object, the resulting object version *depends* on the writing transaction [23]. When a transaction aborts and goes through the undo logic, all its writes are undone. These cause the objects to get new-new versions.

To depict the dependency among two or more transactions we can use the *dependency graph*. The dependency graph depicts three basic dependencies (READ→WRITE, WRITE→READ, WRITE→WRITE). For illustration of all three cases see Figure 2.1.

The most important result of isolation theorems (they can be found in Section 2.1.4) is that any dependency graph without cycles implies an isolated execution of the transaction. If the dependency graph has no cycles, then the transactions' dependency graph can be topologically sorted to make an equivalent execution history in which each transaction ran serially. On the other hand, if there is a cycle then such a sort is impossible to do, because there exists at least two transactions, such that T1 runs before T2, and that T2 runs before T1 [23].



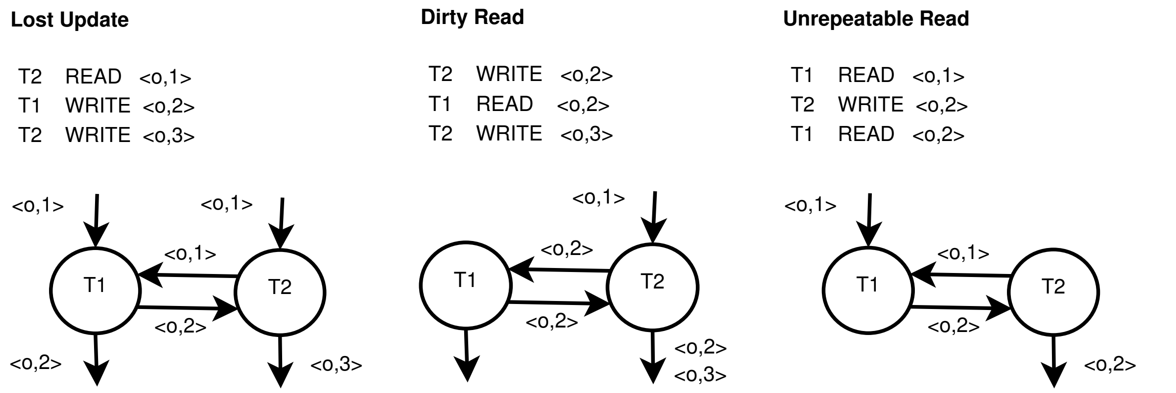
(a) READ→WRITE dependency. (b) WRITE→READ dependency.

(c) WRITE→WRITE dependency.

Figure 2.1: The three cases of transaction dependencies.

2.1.3.3 The Bad Dependencies

Isolation of concurrent running transactions can be violated in various ways. We distinguish three kinds of isolation violation caused by "bad dependencies" called by Gray [23]: *lost* *update, dirty read and unrepeatable read.* These "bad dependencies" can be easily detected in dependency graph, because each of them forms a cycle, see Figure 2.2.



(a) WRITE→WRITE can cause Lost Update. (b) WRITE→READ can cause Dirty Read. (c) READ→WRITE can cause Unrepeatable Read.

Figure 2.2: The three bad transaction dependencies.

Lost Update. Transaction T1's write is ignored by transaction T2, which writes object o based on the original value $< o, 1 >$. A READ-WRITE-WRITE sequence is depicted in the diagram, but a WRITE-WRITE-WRITE sequence forms the same graph.

Example 2.1.1. Lost Update. Two programmers are working on the same program. Each of them made a copy of the program from the repository and worked on this copy independently. After the work is done each of them will copy his version of the program back to the repository. The result is unpredictable. The changes made by the first or by the second programmer will be lost. The resulting version depends on the order of copying.

Dirty Read. T1 reads an object previously written by transaction T2, after that transaction T2 will make changes to the object o. The problem is that the version read by T1 may be inconsistent, because it is not the final (committed) version of o produced by T2.

Example 2.1.2. Dirty Read. With respect to the programming example this situation can be described as follows. If the first programmer pushes the incomplete version of his program into the repository and the second programmer uses this program version for the work and finally the first programmer will push the final version. Hence, the second programmer finished the work using the program version that is not the final.

Unrepeatable Read. T1 reads an object twice, once before transaction T2 updates it and once after committed transaction T2 has updated it. The two read operations return different values for the same object during the same transaction.

Example 2.1.3. Unrepeatable Read. This can be illustrated on the situation when the first programmer uses the version 1 from the repository and in meanwhile the second program installs the version 2 of the program into the repository. When the first programmer is reading the program from the repository then he gets the version 2. So, his first read was unrepeatable.

2.1.4 Isolation Theorems

Isolation theorems are important findings of the transaction theory. In this section we provide a mathematical definition of transaction isolation in terms of execution histories and dependency graphs.

2.1.4.1 Well–Formed and Two–Phased Transactions

Definition 2.1.2. *A transaction is well-formed [23] if each READ, WRITE, and UNLOCK action is covered by a corresponding lock, and all locks are released by the end of the transaction.*

Definition 2.1.3. *A transaction is defined as two-phase [23] if all its LOCK actions precede all its UNLOCK actions. A two-phase transaction T has a growing-phase, T[1], ..., T[j], during which it acquires locks, and a shrinking phase, T[j+1], ..., T[n], during which it releases locks.*

Definition 2.1.4. *A transaction is defined as strict two-phase [23] if all its LOCK actions precede all its UNLOCK actions. A two-phase transaction T has a growing-phase, T[1], ..., T[j], during which it acquires locks, and a shrinking phase, T[j+1], ..., T[n], during which it releases locks. But shrinking phase is done at during COMMIT or ABORT.*

Along the thesis we assume well-formed and two-phase transactions under the term *transaction.*

2.1.4.2 Histories

Definition 2.1.5. *Any sequence-preserving merge of the actions of a set of transactions into a single sequence is called a history [23]*[1] *for the set of transactions and is denoted* $H =<< t, a, o >_i \mid i = 1, ..., n >$.

The simplest histories first run all the actions of one transaction, then run all the actions of another transaction, and so on. Such a transaction at a time histories are called *serial histories.*

We use *serial histories* to define serializability that is a important "feature" of transaction processing.

2.1.4.3 Serializability and Two–Phase locking

Definition 2.1.6. *A* serializable history *over a set S of committed transactions is a history whose effect on any consistent database instance is guaranteed to be equivalent to some* serial history *over S. [50]*

A serializable history can be achieved many ways. The simplest way is to run transactions serially, but this way does not provide high transaction throughput because other transactions has to wait on COMMIT of the currently running transaction. The much more effective way is a controlled interleaving of transactions' operations. This can be done if we use LOCK operation before each READ and WRITE operation. We call this mechanism a locking protocol [50, 23][2].

We also know that if a lock protocol is two–phase then the execution of transactions forms a serial history. We can form this assertion into the following theorem:

Theorem 2.1.1. *Two-phase locking theorem : If all transactions in an execution are two-phase locked, then the execution is serializable. [4]*

For the proof of this theorem, see [4].

[1] Ramakrishnan and Gehrke [50] use another terminology. They call this sequence a *schedule.*

[2] There are also other approaches, which belongs to the category of optimistic methods for concurrency control [39].

2.1.4.4 Lock Compatibility

A history should not complete a lock action on an object while that object is locked by another transaction in an incompatible mode [23]. In other words, locking constrains the set of all allowed histories. Histories that respect the locking constraints are called *legal* [23]. Gray and Reuter are defining legal histories more formally [23]:

Definition 2.1.7. *Transaction t has object o locked in SHARED mode at step k of history H, if for some* $i < k$*, action* $H[i] =< t, SLOCK, o >$*, and if there is no* $< t, UNLOCK, o >$ *action in the subhistory* $H[i+1], ..., H[k-1]$*. Locking in EXCLUSIVE mode at step K is defined analogously.*

Lock compatibility is usually defined by a *compatibility matrix*. The compatibility matrix of simple locking protocol is shown in Table 2.1.

Compatibility		**Mode of Lock**	
		Share	Exclusive
Mode of Request	Share	+	-
	Exclusive	-	-

Table 2.1: A compatibility matrix

Figure 2.3 shows three examples of histories. The first history is legal and serial, firstly transaction T1 is executed and then transaction T2, this history conforms locking protocol constraints. The second history is legal but not serial, operations of transaction T1 and transaction T2 are interleaved, this history also conforms locking protocol constraints. The third history is not legal and not serial because operations of transaction T1 and transaction T2 are interleaved (not serial) and the transaction T2 locks object B in incompatible mode (bold operation in Figure 2.3(c)).

2.1.4.5 Dependency and Wormholes

Definition 2.1.8. *Transaction T1 is said to depend on another transaction T2 in a history H if T1 reads or writes data previously written by T2 in the history H, or if T1 writes an object previously read by T2.*

The previous definition defines dependency property between two transactions. Therefore we can build a *dependency graph*. The dependency graph is a labeled, directed graph in

T1	SLOCK	A
T1	XLOCK	B
T1	READ	A
T1	WRITE	B
T1	UNLOCK	A
T1	UNLOCK	B
T2	SLOCK	A
T2	READ	A
T2	XLOCK	B
T2	WRITE	B
T2	WRITE	B
T2	UNLOCK	A
T2	UNLOCK	B

(a) Legal and serial history.

T2	SLOCK	A
T1	SLOCK	A
T2	READ	A
T2	XLOCK	B
T2	WRITE	B
T2	WRITE	B
T2	UNLOCK	A
T2	UNLOCK	B
T1	XLOCK	B
T1	READ	A
T1	WRITE	B
T1	UNLOCK	A
T1	UNLOCK	B

(b) Legal and not serial history.

T1	SLOCK	A
T1	XLOCK	B
T2	SLOCK	A
T2	READ	A
T2	**XLOCK**	**B**
T2	WRITE	B
T2	WRITE	B
T2	UNLOCK	A
T2	UNLOCK	B
T1	READ	A
T1	WRITE	B
T1	UNLOCK	A
T1	UNLOCK	B

(c) Not legal and not serial history.

Figure 2.3: The example of three execution histories.

which the nodes are transactions and the edges are transaction dependencies labeled with the object versions being read or written by the transactions [23].

Firstly, we define the *object version*:

Definition 2.1.9. *The version of an object o at step k of a history H is an integer and is denoted V(o,k). Initially, each object has version zero (V(o,0)=0). At step k of history H, object o has a version equal to the number of writes of that object before this step:*

$$V(o,k) = ||< t_j, a_j, o_j >\in H | j < k \, and \, a_j = WRITE \, and \, o_j = o||,$$

where || || is the set cardinality function.

Each history H for a set of transactions T_i defines a ternary dependency relation $DEP(H)$, defined as follows:

Definition 2.1.10. *Let* T1 *and* T2 *be any two distinct transactions, let* o *be any object, and let* i,j *be any two steps of* H *with* $i < j$. *Suppose step* H[i] *involves action* a1 *of* T1 *on object* o, *step* H[j] *involves action* a2 *of* T2 *on object* o, *and suppose there is no WRITE* *action of* o *by any transaction between these steps. Then* DEP(H) *is defined as:*

$$< T1, < o, V(o,j) >, T2 > \in DEP(H) \begin{cases} \textit{if a1 is a WRITE and a2 is a WRITE} \\ \textit{if a1 is a WRITE and a2 is a READ} \\ \textit{if a1 is a READ and a2 is a WRITE} \end{cases}$$

The dependency graph for each of $READ \rightarrow WRITE$, $WRITE \rightarrow WRITE$, and $WRITE \rightarrow READ$ dependencies is shown in Figure 2.2.

We can say that two histories for the same set of transactions are *equivalent* if they have the same dependency relation (DEP(H) = DEP(H')). A history is *isolated* if it is equivalent to a *serial history* [23]. The next important finding is that dependencies of a history define a time order of transactions.

Definition 2.1.11. *The time ordering* $<<<_H$ *of the transactions in a history H is the smallest relation satisfying the equation:*

$$\begin{aligned} &T <<<_H T' if < T, o, T' > \in DEP(H)\ for\ some\ object\ version\ o, or \\ &(T <<<_H T'' and < T'', o, T' > \in DEP(H)\ for\ some\ transaction\ T'',\ and\ object\ o). \end{aligned}$$

In other words, $T <<< T'$ if there exists a path in the dependency graph from transaction T to transaction T'. We can use this relation to define functions $BEFORE(T)$ and $AFTER(T)$ that returns the set of all transactions that run before T, or after T respectively. A formal definition is:

Definition 2.1.12.

$$\begin{aligned} BEFORE(T) &= \{T' | T' <<< T\} \\ AFTER(T) &= \{T' | T <<< T'\} \end{aligned}$$

When transaction T is running alone in the database then $BEFORE(T)$ and $AFTER(T)$ sets are empty. In this case, this transaction can be scheduled any way. It does not depend on any other transaction. Even more interesting is when the BEFORE and AFTER sets of T are not empty. The next special case is when BEFORE and AFTER sets are both nonempty. More formally:

Definition 2.1.13. *Transaction* $T' \in BEFORE(T) \cap AFTER(T)$ *then* T' *is called a* *wormhole transaction.*

Transaction T' runs before T and after T simultaneously. Wormhole transactions (all transactions that satisfy previous definition) are named after the points near black holes that reputedly let one travel arbitrarily in time and space [23]. They get this name because they perform actions before T completes and after T completes.

The great finding is that serial histories do not have wormholes. In a serial history, all the actions of one transaction precede actions of another transaction; the first transaction cannot depend on the outputs of the second.

2.1.5 Degrees of Isolation

In previous subsections we consider "Fully isolated" transactions that conform to ACID properties. But this kind of transactions is not needed for most applications. The main motivation for relaxing of isolation property is in increasing transaction throughput. We recognize following degrees of isolation that relax ACID isolation property [23]:

- Degree 0. Transaction is degree 0 isolated if it does not overwrite "dirty data" of another transaction T. T degree of isolation > 0.
- Degree 1. Transaction is degree 1 isolated if it does not contain lost updates.
- Degree 2. Transaction is degree 2 isolated if it does not contain lost updates and dirty reads.
- Degree 3. Transaction is degree 3 isolated if it does not contain lost updates, dirty reads, and repeatable reads are enabled. This degree conforms with ACID properties.

The lock protocols that can ensure degrees of isolation mentioned above are [23]:

- Degree 0. Lock protocol is well-formed with respect to writes.
- Degree 1. Lock protocol is two-phase with respect to exclusive locks and well-formed with respect to writes.
- Degree 2. Lock protocol is two-phase with respect to exclusive locks and well-formed.
- Degree 3. Lock protocol is two-phase and well-formed.

Theorem 2.1.2. *Degrees of isolation theorem: If a transaction observes the degree 0, 1, 2, 3 lock protocol, then any legal history will give that transaction degree 1, 2, 3 isolation, as long as other transactions are at least degree 1.*

For the proof of the theorem see [22].

2.1.5.1 Phantoms

In previous sections we assumed READ and WRITE operations called for objects stored in a database. There exists another problem when we define new operation INSERT, which inserts new object into a database. According to previous sections it can be viewed as a special case of unrepeatable read called *phantom read*. We present the following example for better explanation (this example is written in the SQL language [35]):

Example 2.1.4. Phantom Read.

```
Transaction 1: SELECT * FROM USERS WHERE SALARY>2000;
Transaction 2: INSERT INTO USERS(NAME,SALARY) VALUES ('MARK', 5000);
Transaction 1: SELECT * FROM USERS WHERE SALARY>2000;
```

Explanation: The result of the second read of users includes also a new user MARK because standard lock protocol can lock only existing objects in the database. But when executing the first query object MARK does not exist yet. The second read is Phantom Read. We have to use some kind of range or predicate locks [20] to avoid it.

3 XML Transactions

In this chapter we describe basic principles of transaction processing used in XML databases. Section *Overview* introduces transaction processing of XML data. A difference between relational and XML data model is described in Section 3.2. Section *Locking Protocols* 3.4 illustrates basic locking protocols used in a relational databases and XML databases respectively.

3.1 Overview

A common requirement for database management systems is a concurrency control. There are four well-known properties for a transactional system known as *ACID* [17]. Transaction is generally a unit of work in a database. ACID properties are independent on a database (logical) model (i.e. it must be kept in all transactional database systems, but under special circumstances we can relax them).

Isolation of transactions in a database system is usually ensured by a locking protocol. Direct application of a locking protocol used in relational databases does not provide high concurrency [28, 60] (i.e. transactions are waiting longer than it is necessary).

We consider only well-formed transactions and serializable histories of update operations [4, 23]. All locking protocols quoted in this thesis satisfy these requirements if not stated otherwise. We call locking protocols for (native) XML databases simply *XML-locking protocols*. All those XML-locking protocols are based on a tree locking protocol presented by Gray in [23]. Hence, XML-locking protocols inherit most of its features, e.g. granularity, two-phase. Protocols described in this thesis consider isolation degree 3 (serializable) [23] if not stated otherwise. It implies well–formed and two–phase transactions.

3.2 Relational Data Model vs. XML Data Model

The major differences between XML data and relational data are according to [33]. We can informally say:

- XML data is hierarchical; relational data is represented in a model of logical relationships.

An XML document contains information about the relationship of data items to each other in the form of the hierarchy. With the relational model, the only types of relationships that can be defined are parent table and dependent table relationships.

- XML data is self-describing; relational data is not.

  An XML document contains not only the data, but also tagging for the data that explains what it is. A single document can have different types of data. With the relational model, the content of the data is defined by its column definition. All data in a column must have the same type of data.

- XML data has inherent ordering; relational data does not.

For an XML document, the order in which data items are specified is assumed to be the order of the data in the document. There is often no other way to specify order within the document. For relational data, the order of the rows is not guaranteed unless you specify an ORDER BY clause on one or more columns.

Sometimes the nature of the data dictates the way in which you store it. For example, if the data is naturally hierarchical and self-describing, you might store it as XML data. Hence, we also need effective methods for concurrent processing of XML data. This processing has to be isolated according to running transactions to avoid unwanted results. The best way to achieve this is to use some kind of locking protocol. The previous chapter gives theoretical background which was built for transaction processing in relational databases. Luckily, the findings of transaction theory can be easily used (or extended in some cases) to form, build and apply it in native XML databases/data processing.

3.3 Definitions

These definitions mostly follow XML:DB API specification [34].[1]

Definition 3.3.1. *Database is a set of collections.*

Definition 3.3.2. *Collection C is a pair* $< name, LDC >$*, where name is a name of collection C and LDC is a list of collections and documents stored in collection C.*

[1] The API specification is obsolete nowadays but the underlying model is still used by many XML database vendors.

Definition 3.3.3. *Transaction T running in database D is a pair $< D, LA >$, where LA is a sequence of actions.*

Definition 3.3.4. *Sequence of actions is a sequence starting with a BEGIN action followed by the combination of:*

- *READ(node)*
- *UPDATE(node, new_value)*
- *INSERT_BEFORE(node, node')*
- *INSERT_AFTER(node, node')*
- *DELETE(node)*
- *LOCK(node, lock_mode)*

ending with a COMMIT or a ROLLBACK action.

3.4 Locking Protocols

XML databases provide two basic approaches to access XML data stored in a database. The first approach is a navigational approach based on DOM model [62] that provides operations for accessing and modifying XML elements. The second approach utilizes XDM model [21] and is based on XPath, XQuery and XQuery Update Facility languages. We can recognize more detailed categories for concurrency control. Byun et al. [10] have analyzed semantics of update operations and apply conflict-detection algorithm to recognize whether update operations can be run concurrently or not. This approach requires DTD or other schema of processed documents. On the other hand, other approaches, for example those presented by Jea and Chen in [36], do not need DTD of stored documents and define semantics of update operations extended by lock acquiring during an execution of an expression.

Actual research in the area of locking protocols is concentrated on both models (DOM and XDM). The DOM model exposes methods for navigational approaching of individual parts of an XML document. Probably the most advanced research in this topic is carried out at the University of Kaiserslautern in Germany [31, 27, 28]. Haustein et al. are working on XTC (XML Transaction Coordinator) Project [29] – a system which implements several different algorithms of transaction processing on XML data. There also exists other papers covering DOM model approach [32].

The second approach that utilizes XDM model defines locking protocols for XPath and XQuery Update Facility expressions [47, 46, 36, 10]. The next Section 3.4.1 is focused on the family of DOM locking protocols. After this section we introduce basic techniques of XDM locking in XDM Locking Protocols.

3.4.1 DOM Locking Protocols

In this section we describe basic principles of DOM Locking Protocols. All protocols presented in this section are inspired by the granularity of locks and uses the tree locking protocol presented by Gray in 1976 [22]. We chose the most advanced representative of this family of protocols called taDOM, which was developed as part of XTC project. XTC project uses extended DOM model *taDOM* for document representation. The structure of taDOM model is shown in Figure 3.1.

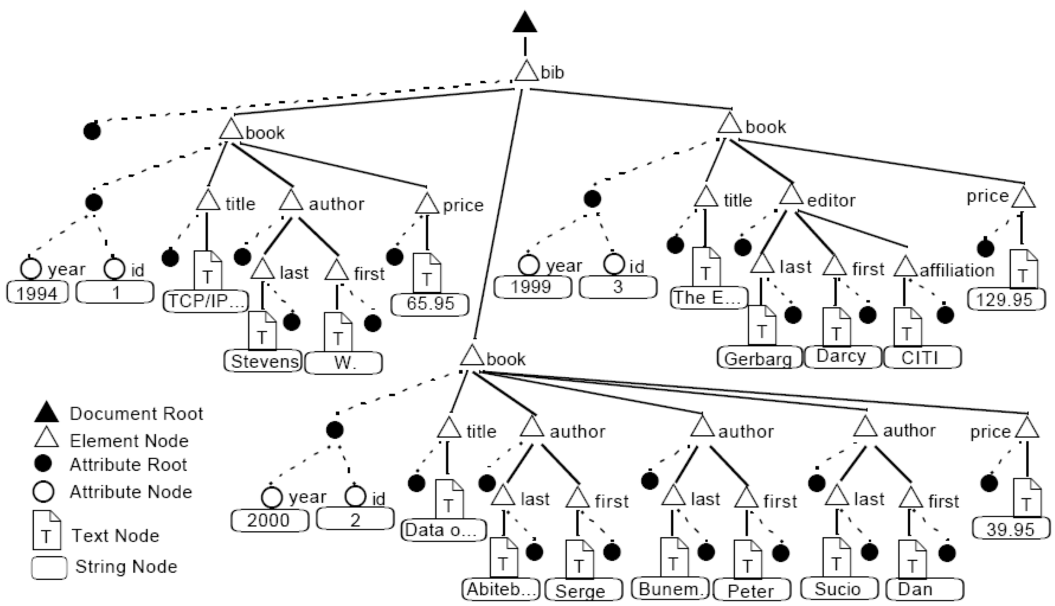


Figure 3.1: The taDOM structure

The first version of the protocol was denominated as taDOM2. Its improved version is then called taDOM2+. Both of these protocols work with DOM Level 2 operations (about 20 methods, see [63]). Next generation of taDOM locking protocols are taDOM3 and

	-	**ER**	**EU**	**EX**
ER	+	+	-	-
EU	+	+	-	-
EX	+	-	-	-

Table 3.1: The Compatibility Matrix of the Edge Locks

$getNode(nodeID)$ **returns** $Node$					
Scenario 0-1 for taDOM3+ Lock Requests:					
Node	Lock	PSE	NSE	FCE	LCE
CN	NR	-	-	-	-

Table 3.2: Lock Scenario for DOM Operation $getNode(nodeID)$

taDOM3+. As expected, these protocols correspond to DOM Level 3 model. The XTC project also provides detailed use cases for these protocols (36 use cases [30]) which completely describe locking scenarios for each operation.

Each of taDOM locking protocols is specified by:

- Compatibility matrix
- Conversion matrix
- Use cases for DOM operations

The compatibility matrix is used when the transaction t_1 is requesting a lock l_1 on a node n and there is a lock l_2 of the transaction t_2. The lock algorithm finds the row l_1 and column l_2 in the compatibility matrix and makes a decision whether to lock (+) or not (-). If the requested lock is incompatible then the transaction is suspended. Table 3.1 describes Compatibility Matrix for edge locks (ER - edge read, EU - edge update and EX - edge exclusive). The compatibility matrix and the conversion matrix can be found in [30].

The conversion matrix is used when the transaction t_1 is requesting a lock l_1 on a node n and there exists a lock l_2 of the same transaction t_1. Lock algorithm finds the row l_1 and column l_2 in the conversion matrix and converts the lock mode of the node. Hence, each transaction has at the most one lock on each node at a time.

Use cases describe semantics of the locking protocol with regard to DOM operations. Table 3.2 contains description of the DOM operation $getNode(nodeID)$. When $getNode(nodeID)$

operation is invoked then the locking mechanism has to put the lock of type NodeRead (NR) on the context node(CN). PSE, NSE, FCE, LCE are abbreviations for previous sibling edge, next sibling edge, first child edge, last child edge. The $getNode(nodeID)$ operation does not put locks on these virtual edges (-).

We consider only taDOM3+ for further research. This protocol is up-to-date nowadays, because it reflects today's needs and was formally checked[2]. The taDOM3+ locking protocol also has low overhead (minimizes access to the storage) [27].

taDOM3+ protocol provides degree 2.99 of isolation [1, 28]. It means that phantom reads[3] are not covered. Therefore it is necessary to do a small extension to these protocols by adding locking of navigation edges to avoid existence of phantom reads. We need to define an additional mechanism – edge locks. To apply edge locks the authors had to extend the XML document model and added new edges between nodes – virtual edges. The compatibility matrix of these locks is more discussed in [28].

3.4.1.1 taDOM Model Structure

The tree-like structure in taDOM is enriched by two new node types: *attributeRoot* and *string* [27]. This representational enhancement does not influence user operations and their semantics on the XML document, but is solely exploited by the lock manager to achieve certain kinds of optimization when the XML document is modified in a cooperative fashion [28].

- *attributeRoot* separates various attribute nodes from their element node. Instead of locking all attribute nodes separately they are locked all together by placing the lock to attributeRoot – concurrency of attribute processing is not allowed.

- A *string* node is attached to the respective text node and only contains the value of this node. It does not allow to block a transaction which only navigates across the node, although a concurrent transaction may have modified the text (content) and may still hold an exclusive lock on it.

[2]Valenta and Siirtola [55] made a formal proof of the protocol correctness. They verified the taDOM locking protocol using model-checking.

[3]Phantom read happens when new data added by a transaction are visible from another transaction.

3.4.1.2 Lock Modes

The taDOM3+ protocol provides a set of lock modes for the nodes as well as for the edges. Edge locks are used to cover phantom reads in an XML document in order to allow desired level of concurrency. The lock modes together with their mutual relationships (expressed as compatibility matrices) provide concurrency and also preserve the expected ACID properties (especially the level of isolation).

```
input:  CN - context node
        LM - lock mode
        t - transaction

lockRequest(CN, LM, t) {                        // request a lock mode
  if(isCompatible(LM, CN.getLock()) {           // if LM is compatible
    lock(CN, LM);                               // assign it
  } else {
    suspend(t);                                 // suspend transaction
    exit();                                     // do not continue
  }
}

getParents(CN, LM) {
  parents:=new Stack();
  while(CN.getParent()!=null){                 // while exists parent
    parents.add(<CN.getParent(), LM.getParentLockType()>);
    CN = CN.getParent();
    LM = LM.getParentLockType();
  }
  return parents;
}

parents:=getParents(CN, LM);
while(!parents.empty()) {
  parent_lock:=parents.pop()
  lockRequest(parent_lock.first(),
              parent_lock.second());            // request lock modes
}
```

Figure 3.2: taDOM Locking Algorithm

3.4.1.3 Locking Protocol Algorithm

The Locking Protocol Algorithm is described in figure 3.2. This algorithm is based on two basic operations:

- *boolean isCompatible(LockType l_1, LockType l_2)* - this function checks compatibility of lock modes l_1 and l_2
- *void lock(Node n, LockType l)* - assigns a lock l for a node n, if there is already assigned a lock mode, then conversion of lock modes is applied using the operation *combineWith*, which implements conversion matrix.

The following example 3.4.1 shows how the locking in taDOM protocol works. taDOM locking protocol is inspired by granular locks that are used for hierarchical locking [23], hence it is important to start locking from the root node to the context node to minimize deadlock probability [23].

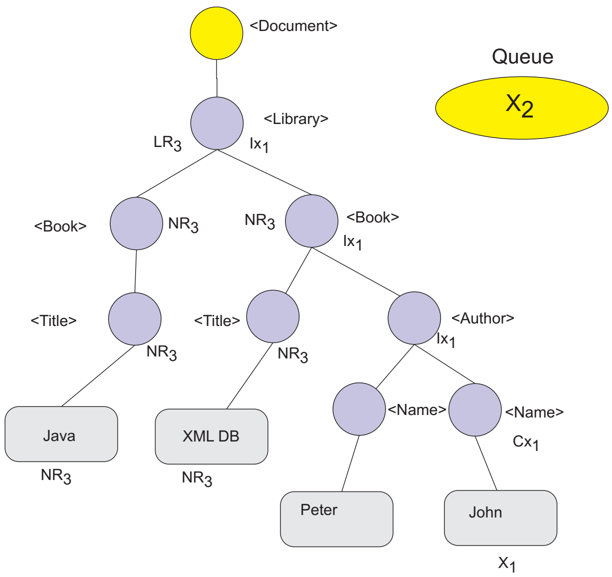


Figure 3.3: Lock Protocol Application Example

Example 3.4.1 (taDOM locking.). *Let us there exists three transactions T_1, T_2, and T_3. Transaction T_1 is updating a text value "John". This text node has to be locked by an exclusive lock. The lock manager assigns a lock mode* `CX` *to the node* `<Name>`*, and to all his predecessors is a lock mode* `IX` *assigned. Simultaneously, the transaction T_2 is going to* *delete a node* `<Author>`*, a lock* `X` *has to be assigned to a node "Peter", but this operation is not allowed because there exists a lock mode* `IX` *on a node* `<Author>`*.*

The request for this lock mode is suspended and is added to a queue. Then this request is waiting for a release of a lock mode on a node `<Author>`*. Simultaneously, the transaction* T_3 *is processing a query which is generating a listing of all books and autors.* T_3 *has to request a lock mode* `LR` *on a node* `<Library>` *to access all direct children of a node* `<Library>`*.* T_3 *has to also acquire a lock mode* `NR` *on all children.*

The previous example was published in [27, 56].

3.4.2 XDM Locking Protocols

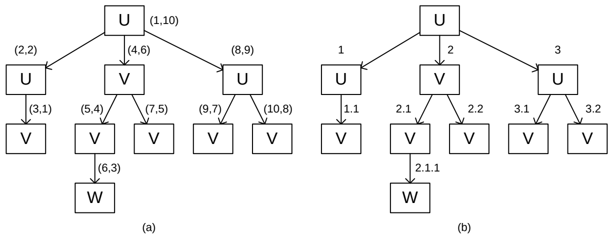
This section describes basic approaches of XDM locking. On the one hand we will present this family of protocols on a representative protocol called XDGL [46, 45], which was developed by Peter Pleshachkov and Sergei Kuznetcov for native XML database Sedna. XDGL protocol uses DataGuide structure. On the other hand we present another approach based on locking of nodes during XPath execution in Subsection 3.4.2.2.

3.4.2.1 XDGL Protocol

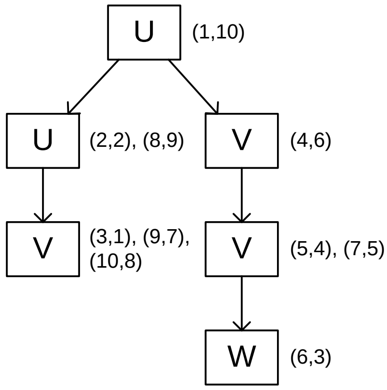
In this section we descibe basic mechanism of XDGL protocol. The most of this section is adopted from the paper by Pleshachkov [46]. If transactions need to lock the same objects, they have to check whether the locks are compatible or not. XDGL protocol requires transaction to follow strict two-phase locking protocol (S2PL). It means according to S2PL a transaction, acquired a lock, keeps it until the end. This protocol is based on the locking of DataGuide indexing structure.

DataGuide was one of the first NXDBMS-specific indexing structures. It allows for indexing structure of XML documents. More specifically, a DataGuide of an XML document is a tree. Its each node represents a single root-to-leaf path of XML node names in the XML document. Its each edge represents that XML nodes on the path represented by the parent are parents of the XML nodes on the path represented by the child. A DataGuide for the sample XML tree is shown in Figure 3.4.

For each of its nodes a DataGuide indexes a sequence of XML nodes on the path represented by the node. For each indexed XML node, the DataGuide indexes the identification number



(a) XML tree with numbering



(b) DataGuide

Figure 3.4: An example of XML tree and the corresponding DataGuide.

assigned to the XML node by the chosen numbering schema. It then allows for providing structural join algorithms with required input streams of XML nodes. In the basic version, XML nodes with a given name are put into a common stream. However, a DataGuide allows for more advanced streaming schemas. For example, it may provide a separate stream for each of its nodes. In other words, XML nodes targeted by the same root-to-leaf path of names are put into a common stream. As shown in [14], this improves the time complexity of structural join algorithms when evaluating twig pattern parent–child edges. It is also possible to reduce the space complexity by stream compression as shown in [3].

Pleshachkov et al. introduced granular locking protocol on DataGuide. The protocol defines intentional locks in addition to shared and exclusive locks. To set a shared lock on an object a transaction T must firstly set an intention locks on its ancestors. But there are a number of use cases when the locking of the entire subtree, as the common granular locking protocol does, is not necessary.

Pleshachkov et al. gave this Use Case to explain it.

Example 3.4.2 (Use Case 1)**.** *Let us suppose that transaction T_1 has issued the XPath query* `/doc/person/name`*. It should be possible for transaction T_2 to insert empty element* `<person/>` *as a child of doc element. According to the granular locking protocol T_1 must lock name subtree while T_2 must lock the entire person subtree including name element. Thus, T_1 and T_2 cannot be executed concurrently.*

In fact the previous Use Case shows that transactions T_1 and T_2 do not conflict. They would conflict if T2 inserted `<person><name>Tanya</name></person>` element inside doc

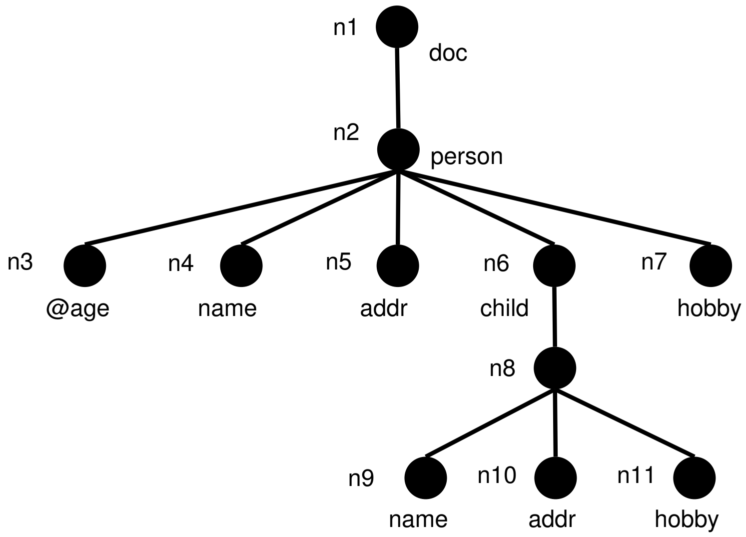


Figure 3.5: DataGuide of the document D

element. To avoid locking of the entire subtree, XDGL use locks on the DataGuide's nodes. This way XDGL can provide [46] high degree of concurrency and, in particular solve the above problem. Besides, XDGL introduce some special shared locks on DataGuides nodes, utilized by insert operations.

To remedy the phantom problem the XDGL protocol introduces special logical locks like the taDOM protocol. They allow to lock name under the DataGuide's node. These locks are useful for such queries as //addr. According to the DTD of document D, person element is defined recursively. Therefore, D's DataGuide in Figure 3.5 could contain random number of the addr nodes. A logical lock on the addr name on the D's DataGuide denies other transactions to insert any element with the name addr.

Logical locks add a great deal of complexity to the XDGL protocol. Hence, at first we will describe a simplified variant of XDGL without logical locks. However, we will note that this variant does not ensure serializability [46].

Simplified XDGL Method. Concurrent operations may result in inconsistent data unless controlled properly. To avoid this kind of problems we must serialize concurrent operations. We employ locks as a mean of synchronization. Let us define the kinds of locks we need.

- SI, SA and SB locks. These special shared locks are used by insert operations. They provide high degree of concurrency that could be achieved because of the insert operator semantics. As we have already mentioned, there are three types of insert operators: insert-into, insert-after and insert-before. Insert-into operator adds a child or an attribute to a node. Insert-after operator creates a sibling for a node. Thus, we add a node to the parent next to our node in the document order [46]. Insert-before operator is defined in a similar way. SI (shared insert), SA (shared after) and SB (shared before) locks block concurrent insert operations of the same type. These locks also protect the parent node. For instance, a transaction cannot delete this node while such a lock is held.

- X lock. The lock sets exclusive mode on a DataGuide node. For instance, this lock is obtained for a newly created node.

- ST lock. The lock sets shared mode on a DataGuides subtree. XPath queries require this kind of locks. Due to the semantics of XPath the results of the location path are the subtrees selected by the last location step. It implies the request of the ST (shared tree) lock for subtrees retrieved by location path.

- XT lock. The lock sets exclusive mode on a DataGuide's subtree. We use it for delete operations. The delete operator drops the subtrees defined by location path. It implies the request of the XT (exclusive tree) locks for these subtrees.

- IS lock. According to the granular locking protocol we have to obtain these locks on each ancestor of the node which is to be locked in a shared mode.

- IX lock. According to the granular locking protocol we have to obtain these locks on each ancestor of the node which is to be locked in an exclusive mode.

	granted							
requested	SI	SA	SB	X	ST	XT	IS	IX
SI	-	+	+	-	+	-	+	+
SA	+	-	+	-	+	-	+	+
SB	+	+	-	-	+	-	+	+
X	-	-	-	-	-	-	+	+
ST	+	+	+	-	+	-	+	-
XT	-	-	-	-	-	-	-	-
IS	+	+	+	+	+	-	+	+
IX	+	+	+	+	-	-	+	+

Table 3.3: XDGL Compatibility Matrix

Table 3.3 shows compatibility matrix for the lock modes defined above. A compatibility matrix indicates whether a lock of mode M1 may be granted to a transaction, while a lock of mode M2 is presently held by another transaction.

Note, that IX and X locks are compatible since IX lock on a node only implies the intention to lock the descendants of the node. But it does not imply the lock on the node itself. SI (SA, SB) lock is not compatible with SI (SA, SB) lock, which prevents concurrent insert-into (insert-after, insert-before) operations upon the same node.

Pleshackov et al. shows that both transactions in the Use Case 1 can proceed with proposed locking method. According to XDGL mechanisms, transaction T_1 must obtain IS lock on nodes n1, n2 and ST lock on node n4. At the same time T_2 must obtain IX lock on n1 and X lock on n2. As all locks are compatible transactions T_1 and T_2 could be executed concurrently.

Logical Locks and XDGL In XPath language we can get nodes at any level of the document using descendant axis. Thus, we should prevent phantom appearance in such queries.

Inserts performed by concurrent transactions are the only source of phantoms. One way to prevent phantoms is to request locks of the coarser granules. It is obvious that this would lead to significant decrease in concurrency [46]. For this reason, the XDGL protocol introduces logical locks. Logical lock (L, node-name) is requested for the name of the DataGuide's node.

For instance, the query `/doc/person//addr` requires logical lock (L, addr) on node n2, as well as delete statement DELETE `//hobby` requires logical lock (L, hobby) on the DataGuides root. In turn, a transaction, which wants to insert new node in the document should obtain (IN, node-name) lock on the all ancestors of the node to be inserted. IN is short for Insert New Node. (IN, node-name1) lock is compatible with (L, node-name2) lock if and only if node-name1 differs from node-name2. Note, that L and IN locks do not conflict with locks introduced in the previous section.

Example 3.4.3 (phantom prevention)**.** *Let us suppose that transaction T_1 retrieves all age attributes found at any level inside person elements which can be found themselves inside doc. In XPath such query looks like this:* `/doc/person//@age`*. At the same time transaction T_2 inserts new age attribute into the person element by the following statement: INSERT attribute{age}{54} INTO* `/doc/person/child/person`*. It is easy to see that the* *second transaction might add a phantom node for the first one. However, our locking rules prevent this situation. (L, @age) lock is not compatible with (IN, @age) lock. Thus, the insertion of the age attribute is denied.*

3.4.2.2 XLP Protocol

In this section we introduce XPath locking protocol based on locking of accessed nodes. This protocol was presented by Jea and Chen in [36]. The difference between XLP and XDGL is in the type of locked nodes. XLP protocol locks nodes of the accessed XML document during the evalution of XPath query. On the other hand, XDGL locks nodes of the DataGuide index structure and is not directly accessing nodes of the XML document.

In XLP [36] there exists five different types of operations when evaluating a location path. The Pass-by operation is used for the Node-Test and Predicate in each location step, while the Read, Write, Insert, and Delete operations are used for processing the destination nodes. According to these operations XLP defines five lock modes, denoted by P-, R-, W-, I- and D-locks, which has to be acquired before the Pass-by, Read, Write, Insert, and Delete operations, respectively.

Definitions

First we have to introduce definitions which were originally given by Jea and Chen in [36] for the purpose of XLP. We need them to make the following text clearer.

The symbols $S_{i,j}$, L_j and l_j are used to model an XPath expression. $S_{i,j}$ denotes the ith location step in location path L_j with length l_j (i.e. number of location steps in L_j). Hence, location path L_j with m location steps can be denoted by $/S_{1,j}/S_{2,j}/S_{3,j}/./S_{m,j}$, where $m = l_j$.

We define the three sets $C(S_{i,j})$, $M(S_{i,j})$ and $R(S_{i,j})$ to model nodes explicitly indicated in an XPath expression. $C(S_{i,j})$ denotes the set of context nodes of $S_{i,j}$. With respect to $C(S_{i,j})$, $M(S_{i,j})$ denotes the set of (mid-result) nodes that satisfy the structural constraint Axis::Node-Test of $S_{i,j}$. On the other hand, $R(S_{i,j})$, the set of result nodes of $S_{i,j}$, is the set of nodes in $M(S_{i,j})$ satisfying the Predicate of $S_{i,j}$. In fact, the result nodes of $S_{i,j}$ become the context nodes of $S_{i+1,j}$. That is, $R(S_{i,j}) = C(S_{i+1,j})$.

Further, we use the symbols $M_I(S_{i,j})$ and $R_I(S_{i,j})$ to denote the sets of nodes not explicitly indicated in location step $S_{i,j}$ but implicitly visited by the query evaluator when navigating $M(S_{i,j})$ and $R(S_{i,j})$, respectively. The nodes in these sets are called the implicit pass-by

Symbol	**Description**
L_j	Location path L_j
$S_{i,j}$	The ith location step in the location path L_j
l_j	Length of the location path L_j
$C(S_{i,j})$	Context nodes in $S_{i,j}$
$M(S_{i,j})$	Mid-result nodes in $S_{i,j}$
$M_I(S_{i,j})$	Implicit pass-by nodes of $M(S_{i,j})$
$R(S_{i,j})$	$= C(S_{i+1,j})$, the set of result nodes in $S_{i,j}$, $R(S_{i,j}) \subseteq M(S_{i,j})$
$R_I(S_{i,j})$	Implicit pass-by nodes of $R(S_{i,j})$, $R_I(S_{i,j}) \subseteq M_I(S_{i,j})$
$N_d(L_j)$	$= R(S_{lj,j})$, the set of destination nodes in L_j

Table 3.4: Symbols

nodes. The nodes included in $M_I(S_{i,j})$ depend on $C(S_{i,j})$, $M(S_{i,j})$ and the axis in $S_{i,j}$. For the preceding, preceding-sibling, following and following-sibling axes of XPath expression, $M_I(S_{i,j})$ includes the nodes in paths starting from the root (/) to the nodes in $M(S_{i,j})$ but excluding the root and the nodes in $M(S_{i,j})$, since nodes in $C(S_{i,j})$ and $M(S_{i,j})$ are in different paths for these axes. For the descendant and descendant-or-self axes, $MI(S_{i,j})$ includes the nodes in paths starting from the nodes in $C(S_{i,j})$ to the nodes in $M(S_{i,j})$, but excluding the nodes in $C(S_{i,j})$ and $M(S_{i,j})$. Moreover, $MI(S_{i,j})$ is an empty set for the self, parent, ancestor, child and ancestor-or-self axes. Note that we treat the attribute axis in the same way as the child axes for their similar access behavior in the XPath model. The set $M_I(S_{i,j}) \cup M(S_{i,j})$ of $S_{i,j}$, i.e. all the nodes visited in $S_{i,j}$, is called the $M-set$ of $S_{i,j}$ for simplicity. Finally we define the set of destination nodes of location path L_j, denoted by $N_d(L_j)$, as the set of result nodes after evaluating path L_j. In fact, $N_d(L_j)$ is equal to $R(S_{lj,j})$, where l_j is the length of L_j.

The previous definitions are summarized in Table 3.4.

Lock Modes

We give semantics of lock modes according to [36]:

- P-lock mode. The P-lock is a shared lock designed for the Pass-by operation. In other words it is intended for mid-results of XPath location path. At the final location step of the path, P-locks on the destination nodes are eventually upgraded to R-, W-, I- or D-locks, depending on the type of operation on the destination nodes. P-locks (for the Pass-by operations) are compatible with R-locks (for the Read operations). They are conditionally compatible with W-, I- and D-locks.

- R-lock mode. The operation $(R(x), L_j)$ in a transaction must acquire R-locks on the destination nodes in the location path L_j. R-locks are upgraded from P-locks. An R-lock (for the Read operations) is compatible with a P-lock (for the Pass-by operations) and an I-lock (for the Insert operations).

- W-lock. The operation $(W(x), L_j)$ in a transaction must acquire W-locks on the destination nodes in the location path L_j. W-locks are upgraded from P-locks. The W-lock (for the Write operations) is compatible with the I-lock (for the Insert operations), but conditionally compatible with the P-lock (for the Pass-by operations), it is incompatible with the R-lock (for the Read operations) and D-lock (for the Delete operations).

- I-lock. The operation $(I(x), L_j)$ must acquire I-locks on the destination nodes in the location path L_j. I-locks are upgraded from P-locks. They are compatible with R- and W-locks (for the Read and Write operations), but incompatible with I-locks and D-locks (for the Insert and Delete operations).

- D-lock. The operation $(D(x), L_j)$ must acquire D-locks on the destination nodes in the location path L_j. D-locks are upgraded from P-locks. When deleting a node, all of its child nodes are also deleted. As a result, D-locks (for the Delete operations) are incompatible with other types of locks except the P-locks.

The compatibility matrix is summarized in Table 3.5. The compatibility of various lock modes in XLP, where an + or - in an entry indicates that the lock modes for the two corresponding operations are compatible or incompatible respectively, and an x indicates that the lock modes for the two corresponding operations are either compatible or incompatible depending on whether the condition $x \notin R(S)$ and $x \notin R_I(S)$ (i.e. the nodes x are sieved out by the Predicate of S) holds for some location step S in location path Lj.

According to [36] the following six rules define XLP.

- **Two-phase Locking Rule.** All lock modes, except P-locks, that are acquired or released must observe the two-phase locking protocol (2PL).

- **P-lock Rule.** Nodes in the M-set of $S_{i,j}$ are all locked by P-locks before performing the Node-Test and Predicate of location step $S_{i,j}$.

- **Granularity Rules.**

	granted				
requested	P	R	W	I	D
P	+	+	x	x	x
R	+	+	-	+	-
W	x	-	-	+	-
I	x	+	+	-	-
D	x	-	-	-	-

Table 3.5: XLP Compatibility Matrix. + compatible. - incompatible. x conditionally compatible

1. Lock granularity of P-, R-, I-, or W-locks on a node is only the node itself.
2. Lock granularity of D-locks on a node includes the whole subtree rooted at the node.

- **Upgrade Rules.**

  1. The P-locks on $N_d(L_j)$ are upgraded to I-locks before inserting nodes into $N_d(L_j)$.
  2. The P-locks on $N_d(L_j)$ are upgraded to R- or W-locks before reading or writing.
  3. The P-lock on a node in $N_d(L_j)$ is upgraded to D-lock before deleting the node only if P-locks on all the nodes in its subtree are acquired; that is, the Granularity Rule (2) is satisfied.

- **Compatibility Rule.** A particular type of lock on location step S_i can be granted as long as the compatibility matrix is respected.

- **Release Rules**

  1. R-, W-, I- or D-locks on $N_d(L_j)$ (i.e. $R(S_{lj,j})$) can only be released in the shrinking phase of a transaction; that is, releasing them must observe the two-phase locking rule.
  2. P-locks on nodes in the set $\{x|x \in R_I(S_{i,j}) \cup R(S_{i,j}) \vee R(S_{lj,j}), i \in [1, l_j]$ for location path $L_j\}$ are released only in the shrinking phase; that is, releasing P-locks on these nodes must observe the Two-phase Locking Rule.
  3. P-locks on $(M_I(S_{i,j}) - R_I(S_{i,j})) \cup (M(S_{i,j}) - R(S_{i,j}))$ are released after location step $S_{i,j}$ finishes.

XML-λ Operation	**DOM Operation**
$0 - ary\ function\ /$	$getDocumentElement()$
$application\ /$	$getChildNodes()$
$projection$	$getTagName(projection)$

Table 3.6: XML-λ Operations to DOM Mapping

3.5 Locking Protocol for a Functional XML Update Language

In this section, we provide the technique for transaction isolation of a functional update language, XML-λ [40], by utilizing taDOM locking protocol described in Section 3.4.1. We published results of this section in [58]. The provided technique is based on a translation of XML-λ statements into DOM API calls using a top-down parser directed by an attributed LL(1) translation grammar. For easier specification of transformation between XML-λ primitives and DOM operations we define new operation $\diamond$:

$$f^{+}(v) = \{f^{1}(v), f^{2}(v), f^{3}(v), \dots\}$$

$$f^{+}(v) \diamond g() = \bigcup_{u=1}^{\infty} \{g(f^{u}(v))\}$$

This operation is defined on sets. We can say that the $g()$ function is applied on each element of a set. The XML-λ language has three main operations for accessing and querying nodes in a document.

Mapping these operations to the taDOM3+ protocol is shown in Table 3.6.

3.5.1 A Pinch of Translation Theory

We solved the problem of mapping by translation from one language to another. The straightforward approach is based on construction of an attributed translation grammar [2]. Then all queries written in XML-λ can be translated into a sequence of DOM operations.

Here we refer shortly to definition related to translation grammars – note that we use an attributed translation grammar, i.e. a context-free grammar augmented with attributes, output symbols and semantic rules. The attributed translation grammar is 4-tuple $APG =<$

$PG, A, V, F >$, where PG is a basic translation grammar $PG =< N, \Sigma, D, R, S >$. N is a finite set of non-terminal symbols, Σ is a set of terminals, D is a set of output symbols, R is a set of grammar rules $A => \alpha$, where $A \in N$, $\alpha \in (N \cup \Sigma \cup D)*$ and S is the start symbol, $S \in N$.

Remaining symbols are related to APG and have the following meaning:

A is a finite set of attributes. It is divided into two disjoint sets for synthetized (denoted $Synth$) and inherited (denoted I) attributes.
V is a mapping that assigns a set of attributes to each non-terminal symbol $X \in N$.
F is a finite set of semantic rules.

The example stated in the following section is based on this formalism.

3.5.2 XML-λ to DOM Translation Grammar

We use the standard formal translation directed by an LL(1) parser where the formal translation is described by translation grammar as follows:

$N = \{S, R_0, R_1, T\}$
$\Sigma = \{/, sL, var\}$
$D = \{$ ⓢ,ⓣ,ⓒ $\}$
$R = \{\ S \rightarrow /\ R_0 | var R_1,$
$\quad R_0 \rightarrow sL$ ⓢ $T\ R_1,$
$\quad R_1 \rightarrow /$ ⓒ $sL\ T\ R_1 | \epsilon,$
$\quad T \rightarrow$ ⓣ $\}$

Note that terminal symbols are output tokens from a lexical analyzer.

We proposed necessary attributes for translation $A = \{name, string\}$, where $I(T) = \{name\}$, $I($ⓣ$) = \{name\}$, $Synth(sL) = \{string\}$. Attributes are used for storing tag names in the process of translation.

Syntax and semantics of the translation grammar is described in Table 3.7.

After translation the output symbols are rewritten in the following way:

$$ⓢ \rightarrow doc \diamond getDocumentElement() \diamond getChildNodes()^{+}$$
$$ⓣ \rightarrow \diamond getTagName(ⓣ.\text{name})$$
$$ⓒ \rightarrow \diamond getChildNodes()$$

Following example shows how we can transform XML-λ queries to DOM operations. These operations implicitly use taDOM3+ locking protocol synchronization primitives.

3.5.3 XML-λ Query Evaluation Example

Let us have a look at an example of a delete operation in the XML-λ language. Following statement deletes all books specified by given title:

```
xmldata("bib.xml")
delete( lambda b ( /book(b) and
          b/title = "TCP/IP Unleashed"))
```

We translate the inner expression of the statement

```
(/book(b) and b/title = "TCP/IP Unleashed")
```

The translation is based on a top-down method using expansion operation $\Rightarrow$. Expansion rule depends on the top terminal of the processed input string. Then we can use a standard LL(1) parser. Translation then starts as follows:

$$S \Rightarrow /\ R_0 \overset{R_0}{\Rightarrow} /\ sL\ ⓢ\ \ T\ R_1 \overset{T}{\Rightarrow}\ /\ sL\ ⓢ\ ⓣ\ R_1 \overset{R_1}{\Rightarrow}\ /\ sL\ ⓢ\ ⓣ$$

By this derivation we have translated the first part of the expression – `/book(b)`.
Then, we continue with the second part:

Syntax	**Semantics**
$S \rightarrow /\ R_0 \vert var R_1$	
$R_0 \rightarrow sL\ ⓢ\ T\ R_1$	T.name := sL.string
$R_1 \rightarrow /\ ⓒ\ sL\ T\ R_1 \vert \epsilon$	T.name := sL.string
$T \rightarrow ⓣ$	ⓣ.name := T.name

Table 3.7: Syntax and Semantics Table

Symbol	Inherited attributes	Synthesized attributes
T	name	
sL		string

Table 3.8: Inherited and Synthesized Attributes of Symbols

$$S \Rightarrow \ var\ R_1 \overset{R_1}{\Rightarrow} var/ \text{ (c) } sLTR_1 \overset{T}{\Rightarrow} var/ \text{ (c) } sL \text{ (t) } R_1 \overset{R_1}{\Rightarrow} var/ \text{ (c) } sL \text{ (t)}$$

We get the translated string by omitting input symbols. We suppose that the semantic rules were applied during translation. In the input symbol `var` we saved the first part of the translation. The second part is concatenated with the first part through the variable `b`. The output of the translation is the following sequence of output symbols: (s) (t) (c) (t).

We can rewrite these output symbols to taDOM operations and then we get:

$$doc \diamond getDocumentElement() \diamond getChildNodes()^{+} \diamond getTagName(\text{(t)}.\text{name})$$
$$\diamond getChildNodes() \diamond getTagName(\text{(t)}.\text{name})$$

The main part of the update statement is the path expression. Now we have to select nodes which satisfy condition – *title* $=$ "*TCP/IP Unleashed*". The string comparison operation is not a DOM operation, so for purpose of this paper is omitted here.

The translation grammar described above can be directly used to ensure isolation of transactions in the XML-λ language.

4 XQuery and XQuery Update Facility

In this chapter, we describe the XQuery Update Facility [12] (XQUF) language which extends the XQuery language by updating constructs. We provide the syntax of the language in Extended Backus-Naur Form (EBNF). The meaning of update constructs is described using denotational semantics. As a formal tool which proves the correctness of the given semantics we used The Maude System [15].

4.1 Concrete Syntax and Semantics

We focus on the concrete syntax of the XQUF language. This language is an extension of the XQuery language. XQUF 1.0 extends the syntax of XQuery by adding five new kinds of expressions, named insert, delete, replace, rename, and transform expressions. The formal semantics of XQuery 1.0 [5] is defined for a minimal subset of the language called XQuery Core [19]. The other language constructs can be normalized into XQuery Core. We assume using XQuery Core semantics for simple expressions mentioned in XQUF. First we introduce XQuery 1.0 and XPath 2.0 transaction semantics based on extension of formal semantics specification given in XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition) [18]. Second we introduce XQUF syntax for new language constructs and in Section 4.6 we provide formal semantics of them. The full syntax of XQUF and XQuery language is listed in Appendix A.

The W3C XQUF specification describes the language semantics using Update operations that modifies the XDM instance. This specification does not consider concurrency issues that arise in transaction processing. We figured out this by extending update operations semantics by the transaction semantics.

4.2 Semantics Definitions

In this section we introduce a notation, sorts and function definitions used in other sections for semantics definitions. First we describe symbols used along the text to allow reader clear understanding of the specification. We use *sort* keyword to denote sorts, *co* keyword denotes constructors of sorts and *op* keyword denotes operations with sorts. The meaning of sort is to differ between sort of data and data type. Sort of data denotes a set of "values" of the same kind (for example natural numbers, or days of the week). On the other hand data type is more complex, it contains a set of "values" together with operations. The following specification uses standard mathematical symbols as $\times$ for cartesian product, $\longrightarrow$ for function operator, $\{x : S\}$ to denote a set which contains elements of type S and $(x : S)$ to denote a list which contains elements of type S. We use two kinds of semantics rules in the text. The first kind is a conditional rule of the form:

$$\frac{B_1 ... B_n}{E_0 = E_1}$$

This rule can be interpreted as equation $E_0 = E_1$ iff all conditions $B_1 ... B_n$ holds. The second kind is an equation of the form:

$$E_0 = E_1$$

This equation can be applied iff an expression contains the *pattern* specified in E_0.

We use four *unnamed* semantics functions $[\![]\!]$ which differ in the Syntax domain:

$$[\![]\!]_{xque} : \mathbf{Synt}_{xque} \times \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$
$$[\![]\!]_{xquf} : \mathbf{Synt}_{xquf} \times \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$
$$[\![]\!]_{axis} : \mathbf{Synt}_{axis} \times \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$
$$[\![]\!]_{pred} : \mathbf{Synt}_{pred} \times \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow (Bool \times Node \longrightarrow Node)$$

Constraints Checker Function

$$CC : \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

$\mathbf{Synt}_{xque}$ is a set of syntax expressions of XQuery, $\mathbf{Synt}_{xquf}$ is a set of syntax expressions of XQuery Update Facility, $\mathbf{Synt}_{axis}$ is a set of syntax expressions of XQuery axes, $\mathbf{Synt}_{pred}$ is a set of syntax expressions of XQuery predicates, $\mathcal{G}_{cont}$ and $\mathcal{L}_{cont}$ represents global and local context respectively. $\mathcal{G}_{cont}$ and $\mathcal{L}_{cont}$ are data types with the following signature and operations:

Global Context

$$
\begin{aligned}
&\textbf{sort } \mathcal{G}_{cont}\\
\textbf{co gcont} &: Database \times (x : PULItem) \times \{x : Transaction\} \times \\
&\quad \times (x : CCLItem) \times \{x : Lock\} \times WFG \longrightarrow \mathcal{G}_{cont}\\
\textbf{op getPUL} &: \mathcal{G}_{cont} \longrightarrow (x : PULItem)\\
\textbf{op setPUL} &: \mathcal{G}_{cont} \times (x : PULItem) \longrightarrow \mathcal{G}_{cont}\\
\textbf{op getCCL} &: \mathcal{G}_{cont} \longrightarrow (x : CCLITEM)\\
\textbf{op setCCL} &: \mathcal{G}_{cont} \times (x : CCLItem) \longrightarrow \mathcal{G}_{cont}\\
\textbf{op getTRANS} &: \mathcal{G}_{cont} \longrightarrow \{x : Transaction\}\\
\textbf{op setTRANS} &: \mathcal{G}_{cont} \times \{x : Transaction\} \longrightarrow \mathcal{G}_{cont}\\
\textbf{op getLOCKS} &: \mathcal{G}_{cont} \longrightarrow \{x : Lock\}\\
\textbf{op setLOCKS} &: \mathcal{G}_{cont} \times \{x : Lock\} \longrightarrow \mathcal{G}_{cont}\\
\textbf{op getWFG} &: \mathcal{G}_{cont} \longrightarrow WFG\\
\textbf{op setWFG} &: \mathcal{G}_{cont} \times WFG \longrightarrow \mathcal{G}_{cont}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{var } d &: Database\\
\textbf{var } p, p2 &: (x : PULItem)\\
\textbf{var } t, t2 &: \{x : Transaction\}\\
\textbf{var } ccl, ccl2 &: (x : CCLItem)\\
\textbf{var } l, l2 &: \{x : Lock\}\\
\textbf{var } wfg &: WFG\\
getPUL(gcont(d, p, t, ccl, l, wfg)) &= p\\
setPUL(gcont(d, p, t, ccl, l, wfg), p2) &= gcont(d, p2, t, ccl, l, wfg)\\
getCCL(gcont(d, p, t, ccl, l, wfg)) &= ccl\\
setCCL(gcont(d, p, t, ccl, l, wfg), ccl2) &= gcont(d, p, t, ccl2, l, wfg)\\
getTRANS(gcont(d, p, t, ccl, l, wfg)) &= ccl\\
setTRANS(gcont(d, p, t, ccl, l, wfg), t2) &= gcont(d, p, t2, ccl, l, wfg)\\
getLOCKS(gcont(d, p, t, ccl, l, wfg)) &= l\\
setLOCKS(gcont(d, p, t, ccl, l, wfg), l2) &= gcont(d, p, t2, ccl, l2, wfg)\\
getWFG(gcont(d, p, t, ccl, l, wfg)) &= wfg\\
setWFG(gcont(d, p, t, ccl, l, wfg), wfg2) &= gcont(d, p, t2, ccl, l2, wfg2)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{sort Lock}\\
&\textbf{co lock} : Node \times LockMode \times Transaction \longrightarrow Lock\\
&\textbf{op addToLocks} : \mathcal{G}_{cont} \times Lock \longrightarrow \mathcal{G}_{cont}\\
&\textbf{op addToWFG} : \mathcal{G}_{cont} \times Transaction \times Transaction \longrightarrow \mathcal{G}_{cont}
\end{aligned}
$$

Constraint Check List

$$
\begin{aligned}
&\textbf{sort CCLItem}\\
&\textbf{co cclitem} : Const \times [\![\textbf{Synt}_{xque}]\!](\mathcal{G}_{cont}, \mathcal{L}_{cont}) \times\\
&\qquad \times [\![\textbf{Synt}_{xque}]\!](\mathcal{G}_{cont}, \mathcal{L}_{cont}) \longrightarrow PULItem
\end{aligned}
$$

Pending Update List

$$\begin{aligned}\textbf{sort PULItem}\\ \textbf{co pulitem} : (\mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}) \longrightarrow PULItem\end{aligned}$$

Wait-For Graph

$$\begin{aligned}\textbf{sort WFG}\\ \textbf{co emptyWFG} &: \longrightarrow WFG\\ \textbf{op addEdge} &: WFG \times Transaction \times Transaction \longrightarrow WFG\\ \textbf{op removeTransaction} &: WFG \times Transaction \longrightarrow WFG\\ \textbf{op deadlock} &: WFG \longrightarrow Bool\end{aligned}$$

$$\begin{aligned}\textbf{sort Transaction}\\ \textbf{co createTrans} : \mathbb{N} \longrightarrow Transaction\end{aligned}$$

Local Context

$$
\begin{aligned}
\textbf{sort } & \mathcal{L}_{cont} \\
\textbf{co lcont} : & \, Transaction \times (x : PULItem) \times \\
& \times \{x : Error\} \times (x : Node) \times (x : Lock) \longrightarrow \mathcal{L}_{cont} \\
\textbf{op getRES} : & \, \mathcal{L}_{cont} \longrightarrow (x : Node) \\
\textbf{op setRES} : & \, \mathcal{L}_{cont} \times (x : Node) \longrightarrow \mathcal{L}_{cont} \\
\textbf{op getERR} : & \, \mathcal{L}_{cont} \longrightarrow \{x : Error\} \\
\textbf{op setERR} : & \, \mathcal{L}_{cont} \times \{x : Error\} \longrightarrow \mathcal{L}_{cont} \\
\textbf{op getNTL} : & \, \mathcal{L}_{cont} \longrightarrow (x : Lock) \\
\textbf{op setNTL} : & \, \mathcal{L}_{cont} \times (x : Lock) \longrightarrow \mathcal{L}_{cont} \\
\textbf{op getTRANS} : & \, \mathcal{L}_{cont} \longrightarrow Transaction \\
\textbf{op setTRANS} : & \, \mathcal{L}_{cont} \times Transaction \longrightarrow \mathcal{L}_{cont} \\
\textbf{op beginTransaction} : & \, \mathcal{G}_{cont} \times \mathcal{L}_{cont} \times Transaction \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont} \\
\textbf{op commitTransaction} : & \, \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont} \\
\textbf{op abortTransaction} : & \, \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}
\end{aligned}
$$

$$\begin{aligned}
\textbf{var } t, t2 &: Transaction \\
\textbf{var } p &: (x : PULItem) \\
\textbf{var } err, err2 &: \{x : Error\} \\
\textbf{var } res, res2 &: (x : Node) \\
\textbf{var } ntl, ntl2 &: (x : Lock) \\
getRES(lcont(t, p, err, res, ntl)) &= res \\
setRES(lcont(t, p, err, res, ntl), res2) &= lcont(t, p, err, res2, ntl) \\
getERR(lcont(t, p, err, res, ntl)) &= err \\
setERR(lcont(t, p, err, res, ntl), err2) &= lcont(t, p, err2, res, ntl) \\
getNTL(lcont(t, p, err, res, ntl)) &= ntl \\
setNTL(lcont(t, p, err, res, ntl), ntl2) &= lcont(t, p, err, res, ntl2) \\
getTRANS(lcont(t, p, err, res, ntl)) &= t \\
setTRANS(lcont(t, p, err, res, ntl), t2) &= lcont(t2, p, err, res, ntl)
\end{aligned}$$

$$\begin{aligned}
&\textbf{sort Error} \\
\textbf{co error} &: String \longrightarrow Error \\
\textbf{op getErrorString} &: Error \longrightarrow String
\end{aligned}$$

Database and XDM

$$
\begin{aligned}
\textbf{sort Item} & \\
\textbf{subsort Node} & \lhd \textbf{Item} \\
\textbf{subsort Document} & \lhd \textbf{Node} \\
\textbf{subsort Element} & \lhd \textbf{Node} \\
\textbf{subsort Text} & \lhd \textbf{Node} \\
\textbf{subsort Attribute} & \lhd \textbf{Node}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{sort Database} & \\
\textbf{co database} & : \{x : Collection\} \longrightarrow Database
\end{aligned}
$$

$$
\begin{aligned}
\textbf{sort Collection} & \\
\textbf{co collection} & : \{x : Document\} \longrightarrow Collection
\end{aligned}
$$

$$
\textbf{co document} : String \times Element
$$

$$
\begin{aligned}
\textbf{sort Element} & \\
\textbf{co element} & : String \times \{x : Attribute\} \times \{y : Node\} \\
\textbf{op children} & : Element \longrightarrow \{x : Node\} \\
\textbf{op parent} & : Element \longrightarrow Node
\end{aligned}
$$

XQuery Functions

$$
\begin{aligned}
\textbf{op fs:item-at} : &\ \mathcal{G}_{cont} \times \mathcal{L}_{cont} \times \mathbb{N} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont} \\
\textbf{op fs:last-item} : &\ \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont} \\
\textbf{op fn:root} : &\ \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont} \\
\textbf{op fs:plus} : &\ \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \\
\textbf{op fs:minus} : &\ \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \\
\textbf{op fs:length} : &\ \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathbb{N}
\end{aligned}
$$

Auxiliary Functions

$$\textbf{op filter} : \mathcal{G}_{cont} \times \mathcal{L}_{cont} \times (Bool \times Sequence \longrightarrow Sequence) \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

4.3 Light-Weight XDM

The specification of XQuery and XQuery Update Facility 1.0 uses the XQuery 1.0 and XPath 2.0 Data Model (XDM) [21] for XML data representation. For our semantics definition we assume Light-Weight XDM that is a subset of XDM. Light-Weight XDM is depicted in Figure 4.1. We also consider subset of XDM operations defined for the model. The semantics of those operations remained unchanged. The full model's specification is listed in Appendix B.

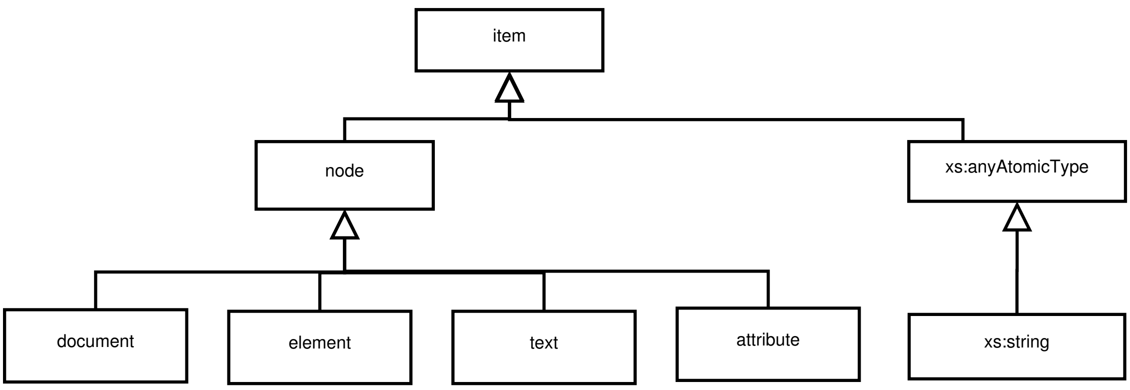


Figure 4.1: Light-Weight XDM

4.4 XQuery and XPath Language Semantics

In this section we provide transaction semantics for XQuery 1.0 and XPath 2.0. This step is needed for correct and complete specification of the semantics of XQUF expressions. The presented transaction semantics of XQUF conforms to isolation level 3 which also needs to lock nodes for reading. Obviously we do not need to specify transaction semantics for XQuery language because all expressions that access data stored in the database must be specified using XPath 2.0 navigational expressions.

XPath 2.0 [11] is an expression language that allows the processing of values conforming to XDM data model defined in [21] . The data model provides a tree representation of XML documents as well as atomic values such as integers, strings, and booleans, and sequences that may contain both references to nodes in an XML document and atomic values. The result of an XPath expression [11] may be a selection of nodes from the input documents, or an atomic value, or more generally, any sequence allowed by the data model. The name of the language derives from its most distinctive feature, the path expression, which provides a mean of hierarchic addressing of the nodes in an XML tree.

The XPath EBNF grammar consists of the highest-level symbol XPath:

```
[1]  XPath       ::= Expr
[2]  Expr        ::= ExprSingle ("," ExprSingle)*
[3]  ExprSingle  ::= ForExpr
                     | QuantifiedExpr
                     | IfExpr
                     | OrExpr
```

The straightforward solution for correct evaluation of XPath expressions according to transaction processing is based on locking of nodes depending on used axes. We introduce transaction semantics only for Core Grammar expressions.

4.4.1 Expression Semantics

The grammar contains four basic types of expressions - ForExpr, QuantifiedExpr, IfExpr and OrExpr. Generally speaking these expressions can access data stored in the database using Path Expressions.

4.4.1.1 Path Expressions

A set representing a syntax domain $\mathbf{Synt}_{xque}$ is generated by Path Expressions' EBNF grammar:

```
[68 (XQuery)]  PathExprXQ  ::=  ("/" RelPathExpr?)
                                | ("//" RelPathExpr)
                                | RelPathExpr
[69 (XQuery)]  RelPathExprXQ  ::=  StepExpr (("/" | "//") StepExpr)*
```

The following semantics equations (semantics function definitions) are inspired by normalization rules from W3C specification [18]. We focused only on a few important kernel functions in our specification. In the W3C specification is the processing described in more detail. We introduce LockRead function, which locks the resulted nodes of the query stored in the local contex variable RES. LockRead function wraps the semantics function $[\![]\!]_{xque}$ by default. In other words we will write $[\![]\!]_{xque}$ instead of $LockRead([\![]\!]_{xque})$. We use [] to access properties of objects, e.g. $l[RES]$ means that we access property RES "stored" in object l. The following rules uses pattern matching. It means that if the left-hand side is satisfied (the expression contains the pattern) then it is rewritten to the expression on the right-hand side. For more details see Section 1.3.

$$\textbf{var } g : \mathcal{G}_{cont}$$

$$\textbf{var } l : \mathcal{L}_{cont}$$

$$\textbf{op LockRead : } \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

$$LockRead(g, l[RES == ()]) = (g, l)$$

$$LockRead(g, l[RES == (Value1)]) = Lock(Value1, SR, g, l)$$

$$LockRead(g,l[RES == (Value1 : Value2)]) =$$
$$= LockRead(Lock(Value1, SR, g, l[RES := RES \setminus (Value1)]))$$

The XQuery (XPath) semantics is following [1]:

$$\textbf{op fn:root :}\ Node \times \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

$$[\![\text{fn:root(self::node)}]\!]_{xque}(g,l) = \text{fn:root(self::node(), g, l)}$$

$$[\![/]\!]_{xque}(g,l) = \text{fn:root(self::node(), g, l)}$$

$$[\![/RelPathExpr]\!]_{xque}(g,l) = [\![\text{fn:root(self::node())}/RelPathExpr]\!](g,l)$$

$$[\![//RelPathExpr]\!]_{xque}(g,l) =$$
$$= [\![\text{fn:root(self::node())/descendant-or-self::node()}/RelPathExpr]\!]_{xque}(g,l)$$

$$[\![RelPathExpr//StepExpr]\!]_{xque}(g,l) =$$
$$= [\![RelPathExpr/\text{descendant-or-self::node()}/StepExpr]\!]_{xque}(g,l)$$

$$[\![RelPathExpr/StepExpr]\!]_{xque}(g,l) = [\![StepExpr]\!]_{xque}([\![RelPathExpr]\!]_{xque}(g,l))$$

Actually the semantics of $[\![StepExpr]\!]_{xque}(g,l)$ is (g',l') and the result is stored inside the local context in the variable RES.

We do not define semantics of functions like *fn:root* or *fs:apply-ordering-mode* here, because it is defined in XQuery specification [18] and we do not redefine its meaning. The only change is that the functions does not operate on immediate parameters but on the result stored in the local context in the variable RES.

4.4.1.2 Steps

A set $\textbf{Synt}_{xque}$ is generated by the Core grammar productions for XPath steps:

```
[46 (Core)]  StepExpr  ::=  PrimaryExpr | AxisStep
```

[1] Function fn:root modifies local state by $l[RES \leftarrow root]$, where $root$ is a root element of the context node.

```
[47 (Core)]  AxisStep  ::=  ReverseStep | ForwardStep
[48 (Core)]  ForwardStep  ::=  ForwardAxis NodeTest
[50 (Core)]  ReverseStep  ::=  ReverseAxis NodeTest
```

If the predicate expression is a numeric literal or the *fn:last* function then the following semantics rules apply:

$$\textbf{var } g : \mathcal{G}_{cont}$$

$$\textbf{var } l : \mathcal{L}_{cont}$$

$$\begin{aligned}&[\![ForwardStep\ PredicateList[NumericLiteral]]\!]_{xque}(g, l) = \\ &\qquad = \text{fs:item-at}([\![ForwardStep\ PredicateList]\!]_{xque}(g, l), NumericLiteral)\end{aligned}$$

$$\begin{aligned}&[\![ForwardStep\ PredicateList[fn : last()]]\!]_{xque}(g, l) = \\ &\qquad = \text{fs:last-item}([\![ForwardStep\ PredicateList]\!]_{xque}(g, l))\end{aligned}$$

And the similar rules apply for the reverse step:

$$\begin{aligned}&[\![ReverseStep\ PredicateList[NumericLiteral]]\!]_{xque}(g, l) = \\ &\qquad = \text{fs:item-at}([\![ReverseStep\ PredicateList]\!]_{xque}(g, l), \\ &\qquad\qquad \text{fs:plus}(1, \text{fs:minus}(\text{fs:length}([\![ReverseStep\ PredicateList]\!]_{xque}(g, l)))))\end{aligned}$$

$$\begin{aligned}&[\![ReverseStep\ PredicateList[fn : last()]]\!]_{xque}(g, l) = \\ &\qquad = \text{fs:item-at}([\![ReverseStepPredicateList]\!]_{xque}(g, l), 1)\end{aligned}$$

When predicates are applied on a forward step, the input sequence is first sorted in document order and duplicates are removed [18]. We do not mention a set of predicates' syntax $\mathbf{Synt}_{pred}$ here. The syntax and semantics of predicates can be found in the specification [18].

$$\llbracket ForwardStep\ PredicateList[Expr] \rrbracket_{xque}(g, l) =$$
$$= \text{filter}(\llbracket ForwardStep\ PredicateList \rrbracket_{xque}(g, l), \llbracket Expr \rrbracket_{pred}(g, l))$$

And the similar rule for the reverse step:

$$\llbracket ReverseStep\ PredicateList[Expr] \rrbracket_{xque}(g, l) =$$
$$= \text{filter}(\llbracket ReverseStep\ PredicateList \rrbracket_{xque}(g, l), \llbracket Expr \rrbracket_{pred}(g, l))$$

Finally, the alone reverse or forward step is processed according to used axis when predicate list is empty:

$$\llbracket ForwardStep \rrbracket_{xque}(g, l) = \llbracket ForwardStep \rrbracket_{axis}(g, l)$$
$$\llbracket ReverseStep \rrbracket_{xque}(g, l) = \llbracket ReverseStep \rrbracket_{axis}(g, l)$$

4.4.1.3 Axes

The Core grammar rules for XPath axes are:

```
[49 (Core)]  ForwardAxis  ::=  ("child" "::")
                               | ("descendant" "::")
                               | ("attribute" "::")
                               | ("self" "::")
                               | ("descendant-or-self" "::")
                               | ("namespace" "::") //not allowed in XQuery
[51 (Core)]  ReverseAxis  ::=  ("parent" "::")
                               | ("ancestor" "::")
                               | ("ancestor-or-self" "::")
```

The previous grammar represents a set of syntax consructs $\mathbf{Synt}_{axis}$. First we define semantics of $ForwardStep$, which is composed of semantics of $ForwardAxis$, $ReverseAxis$

and $NodeTest$.

$$\textbf{op node-test: } (\mathit{Item} \longrightarrow \mathit{Bool}) \times \mathit{Sequence} \times \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

$$\textbf{var } g : \mathcal{G}_{cont}$$

$$\textbf{var } l : \mathcal{L}_{cont}$$

$$\begin{aligned}
[\![ForwardAxis\ NodeTest]\!]_{axis}(g,l) &= \text{node-test}([\![NodeTest]\!]_{ntest}(g,l), \\
&\qquad [\![ForwardAxis]\!]_{axis}(g,l)) \\
[\![ReverseAxis\ NodeTest]\!]_{axis}(g,l) &= \text{node-test}([\![NodeTest]\!]_{ntest}(g,l), \\
&\qquad [\![ReverseAxis]\!]_{axis}(g,l))
\end{aligned}$$

In the following rules we define a function *axis(axis-name, node-sequence, g, l)*. This function returns a pair (g, l), where $l[RES]$ contains output sequence of nodes conforming the selected axis on the *node-sequence*. The first set of rules processes the axis judgement on each individual node in the input *node-sequence*.

$$\begin{aligned}
Axis = \{&\text{self::}, \text{child::}, \text{attribute::}, \text{parent::}, \text{descendant::}, \text{descendant-or-self::}, \\
&\text{ancestor::}, \text{ancestor-or-self}\}
\end{aligned}$$

$$\textbf{op axis: } \mathit{Axis} \times \mathit{Sequence} \times \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

First, we define a rule which initializes the *axis* function for current *node-sequence*. That *node-sequence* is stored in $l[RES]$ and is the result of the previous *Forward or Reverse Step*.

$$axis(a \in Axis, \epsilon, g, l) = axis(a, l[RES], g, l[RES := ()])$$

$$axis(a \in Axis, (), g, l) = (g, l)$$

$$axis(a \in Axis, (Value1, Value2), g, l) = axis(a, Value2, axis(a, Value1, g, l))$$

The *self* axis applied to a NodeValue returns a NodeValue, where a NodeValue represents the context node.

$$axis(\text{self::}, NodeValue, g, l) = Lock(NodeValue, P, g, l[RES \leftarrow (NodeValue)])$$

The child, parent and attribute axis are specified as follows. The element function represents element node structure with attributes and an element value. The element value is an element structure that can contain a single value item or multiple values ordered in a sequence. We did a slight modification of the specification according to semantics needs. For more details about original semantics see XQuery Formal Semantics [18].

$$\begin{aligned} &axis(\text{child::}, element(ElName, \{AttrValue, ElValue\}), g, l) = \\ &\qquad = Lock(ElValue, P, g, l[RES \leftarrow (ElValue)]) \end{aligned}$$

$$\begin{aligned} &axis(\text{attribute::}, element(ElName, \{AttrValue, ElValue\}), g, l) = \\ &\qquad = Lock(AttrValue, P, g, l[RES \leftarrow (AttrValue)]) \end{aligned}$$

$$\begin{aligned} &axis(\text{parent::}, NodeValue, g, l) = Lock(\text{dm:parent}(NodeValue), P, g, \\ &\qquad l[RES \leftarrow (\text{dm:parent}(NodeValue))]) \end{aligned}$$

The *descendant*, *descendant-or-self*, *ancestor*, and *ancestor-or-self* axis are implemented using recursive application of the children and parent axes.

$$\frac{\begin{array}{c} axis(\text{child::}, NodeValue, g, l) = Lock(Value1, SR, g, l[RES \leftarrow (Value1)] \\ axis(\text{descendant::}, Value1, g, l) = Lock(Value2, SR, g, l[RES \leftarrow (Value2)] \end{array}}{\begin{array}{l} axis(\text{descendant::}, NodeValue, g, l) = \\ \quad = Lock((Value1, Value2), P, g, l[RES \leftarrow (Value1, Value2)]) \end{array}}$$

$$\frac{\begin{array}{c} axis(\text{self::}, NodeValue, g, l) = Lock(Value1, P, g, l[RES \leftarrow (Value1)]) \\ axis(\text{descendant::}, Value1, g, l) = Lock(Value2, P, g, l[RES \leftarrow (Value2)]) \end{array}}{\begin{array}{l} axis(\text{descendant-or-self::}, NodeValue, g, l) = \\ \quad = Lock((Value1, Value2), P, g, l[RES \leftarrow (Value1, Value2)]) \end{array}}$$

$$\frac{\begin{array}{r} axis(\text{parent::}, NodeValue, g, l) = Lock(Value1, P, g, l[RES \leftarrow (Value1)]) \\ axis(\text{ancestor::}, Value1, g, l) = Lock(Value2, P, g, l[RES \leftarrow (Value2)]) \end{array}}{\begin{array}{l} axis(\text{ancestor::}, NodeValue, g, l) = \\ \quad = Lock((Value1, Value2), P, g, l[RES \leftarrow (Value1, Value2)]) \end{array}}$$

$$\frac{\begin{array}{r} axis(\text{self::}, NodeValue, g, l) = Lock(Value1, P, g, l[RES \leftarrow (Value1)]) \\ axis(\text{ancestor::}, Value1, g, l) = Lock(Value2, P, g, l[RES \leftarrow (Value2)]) \end{array}}{\begin{array}{l} axis(\text{ancestor-or-self::}, NodeValue, g, l) = \\ \quad = Lock((Value1, Value2), P, g, l[RES \leftarrow (Value1, Value2)]) \end{array}}$$

In all other cases the following rule holds.

$$\frac{Otherwise}{axis(Axis, NodeValue, g, l) = (g, l)}$$

4.4.1.4 Conclusions

We introduced transaction semantics of XPath expressions in this section. Despite the (relative) complexity of XPath grammar the mechanism of locking of axes is good enough to ensure safe transaction processing of all XQuery and XPath expressions, because all expressions finally uses axes to access underlying data. There is no other mechanism to access data stored in a database. Thus, the *axis* function is the key point where to acquire locks on accessed nodes.

4.5 XQuery Update Facility Language Syntax

XQUF 1.0 extends the XQuery's syntax by adding five new kinds of expressions - InsertExpr, DeleteExpr, RenameExpr, ReplaceExpr and TransformExpr. Numbers mentioned in [] refer to the original numbering used in W3C XQuery Update Facility Specification [12].

ExprSingle expression:

```
[32]  ExprSingle ::= FLWORExpr
                   | QuantifiedExpr
                   | TypeswitchExpr
                   | IfExpr
                   | InsertExpr
                   | DeleteExpr
```

```
                    | RenameExpr
                    | ReplaceExpr
                    | TransformExpr
                    | OrExpr
```

Insert expression:

```
[143]  InsertExpr ::= "insert" ("node" | "nodes") SourceExpr
                      InsertExprTargetChoice TargetExpr
[142]  InsertExprTargetChoice ::= (("as" ("first" | "last"))? "into")
                                  | "after"
                                  | "before"
[147]  SourceExpr ::= ExprSingle
[148]  TargetExpr ::= ExprSingle
```

An insert expression is an updating expression [2] that inserts copies of zero or more nodes into a designated position with respect to a target node [12]. The keywords node and nodes may be used interchangeably, regardless of how many nodes are actually inserted [12].

Delete expression:

```
[144]  DeleteExpr ::= "delete" ("node" | "nodes") TargetExpr
[148]  TargetExpr ::= ExprSingle
```

A delete expression deletes zero or more nodes from an XDM instance [12]. XDM is XQuery Data Model described in [21]. The keywords node and nodes may be used interchangeably, regardless of how many nodes are actually deleted. A delete expression is an updating expression [12].

Replace expression:

```
[145]  ReplaceExpr ::= "replace" ("value" "of")? "node" TargetExpr
                                                "with" ExprSingle
[148]  TargetExpr ::= ExprSingle
```

[2]The W3C XQUF recommendation defines the term updating expression as the expression that manipulates XDM.

A replace expression is an updating expression [12]. A replace expression has two forms, depending on whether *value of* is specified [12].

Rename expression:

```
[146]  RenameExpr ::= "rename" "node" TargetExpr "as" NewNameExpr
[148]  TargetExpr ::= ExprSingle
[149]  NewNameExpr ::= ExprSingle
```

A rename expression replaces the name property of a data model node with a new QName. A rename expression is an updating expression.

Transform expression:

```
[150]  TransformExpr ::= "copy" VarName ":=" ExprSingle
                         ("," " VarName ":=" ExprSingle)*
                         "modify" ExprSingle "return" ExprSingle
```

A transform expression can be used to create modified copies of existing nodes in an XDM instance. Each node created by a transform expression has a new node identity. The result of a transform expression is an XDM instance that may include both nodes that were created by the transform expression and other, previously existing nodes. A transform expression is a simple expression because it does not modify the value of any existing nodes [12].

4.6 XQuery Update Facility Language's Semantics

In this section we introduce formal semantics of XQUF. We use the symbol $[\![.]\!]_{xquf}$ for the semantics function of XQUF. First we give a formal semantics of XQUF syntax constructs described in Section 4.5 according to transaction processing. Second we provide formal semantics of update operations. We use the following notation which can be easily mapped to formal definitions given in Section 4.2. The motivation for changing the notation was to provide a better readability of definitions and equations. *PUL* stays for a Pending Update List of update operations. In fact it is an ordered collection. *TRANS* is a set of running transactions and *XDM* is the database instance representing data and the structure of the underlying database. The complete execution flow is described in Section 4.6.2. We abbreviate names of update operations in semantics expressions according to Table 4.2. In XQUF semantics each state g implicitly contains an instance of a Light-Weight XDM Model stored in variable named *XDM* representing underlying database. This variable is omitted in the following rules if it is not needed for better readability.

The semantics evaluation starts with the global state g:

$$g = gcont(XDM, (), \emptyset, (), \emptyset, emptyWFG)$$

and the empty local state l. The local state l is created by the BEGIN expression in the beginning of each transaction. The global state is shared among all transactions running inside the system. On the other hand the local state is owned by the only one transaction (owner) and obviously only the owner can access and change its variables. This behavior is defined in semantics of *BEGIN* expression. To access and modify variables contained in a state we use this notation:

$$g[PUL \leftarrow \text{u:iAttrs}([\![TEN]\!](g, l))]$$

In the previous expression the symbol $\leftarrow$ is used as an infix operator to *add* an *operation* (in that case u:iAttrs) into the set identified by the variable PUL.

Abbreviated Name	Operation Name
u:del	upd:delete
u:iAttrs	upd:insertAttributes
u:iIAL	upd:insertIntoAsLast
u:iIAF	upd:insertIntoAsFirst
u:iI	upd:insertInto
u:iB	upd:insertBefore
u:iA	upd:insertAfter
u:rN	upd:replaceNode
u:ren	upd:rename
u:rEC	upd:replaceElementContent
u:rV	upd:replaceValue

Figure 4.2: Update Operations Names

4.6.1 Expressions' Semantics

In this section we define semantics of each individiual statement. The semantics definition is composed of our semantics and the original semantics considering [12].

4.6.1.1 Insert Expression

$$
\begin{aligned}
&[\![insert\, node\, SEN\; as\; first\; into\, TEN]\!]_{xquf}(g,l) = \\
&\quad = CC(g[PUL \leftarrow \{\text{u:iAttrs}([\![TEN]\!]_{xque}(g,l), AL([\![SEN]\!]_{xque}(g,l))), \\
&\qquad\qquad \text{u:iAF}([\![TEN]\!]_{xque}(g,l), CL([\![SEN]\!]_{xque}(g,l)))\}, \\
&\qquad CCL \leftarrow \{<insertFI, [\![TEN]\!]_{xque}(g,l), [\![SEN]\!]_{xque}(g,l)>\}], l) \\
&[\![insert\, node\, SEN\; as\; last\; into\, TEN]\!]_{xquf}(g,l) = \\
&\quad = CC(g[PUL \leftarrow \{\text{u:iAttrs}([\![TEN]\!]_{xque}(g,l), AL([\![SEN]\!]_{xque}(g,l))), \\
&\qquad\qquad \text{u:iAL}([\![TEN]\!]_{xque}(g,l), CL([\![SEN]\!]_{xque}(g,l)))\}, \\
&\qquad CCL \leftarrow \{<insertLI, [\![TEN]\!]_{xque}(g,l), [\![SEN]\!]_{xque}(g,l)>\}], l) \\
&[\![insert\, node\, SEN\; into\, TEN]\!]_{xquf}(g,l) = \\
&\quad = CC(g[PUL \leftarrow \{\text{u:iAttrs}([\![TEN]\!]_{xque}(g,l), AL([\![SEN]\!]_{xque}(g,l))), \\
&\qquad\qquad \text{u:iI}([\![TEN]\!]_{xque}(g,l), CL([\![SEN]\!]_{xque}(g,l)))\}, \\
&\qquad CCL \leftarrow \{<insertI, [\![TEN]\!]_{xque}(g,l), [\![SEN]\!]_{xque}(g,l)>\}], l) \\
&[\![insert\, node\, SEN\; before\, TEN]\!]_{xquf}(g,l) = \\
&\quad = CC(g[PUL \leftarrow \{\text{u:iAttrs}([\![TEN]\!]_{xque}(g,l), AL([\![SEN]\!]_{xque}(g,l))), \\
&\qquad\qquad \text{u:iB}([\![TEN]\!]_{xque}(g,l), CL([\![SEN]\!]_{xque}(g,l)))\}, \\
&\qquad CCL \leftarrow \{<insertB, [\![TEN]\!]_{xque}(g,l), [\![SEN]\!]_{xque}(g,l)>\}], l) \\
&[\![insert\, node\, SEN\; after\, TEN]\!]_{xquf}(g,l) = \\
&\quad = CC(g[PUL \leftarrow \{\text{u:iAttrs}([\![TEN]\!]_{xque}(g,l), AL([\![SEN]\!]_{xque}(g,l))), \\
&\qquad\qquad \text{u:iA}([\![TEN]\!]_{xque}(g,l), CL([\![SEN]\!]_{xque}(g,l)))\}, \\
&\qquad CCL \leftarrow \{<insertA, [\![TEN]\!]_{xque}(g,l), [\![SEN]\!]_{xque}(g,l)>\}], l)
\end{aligned}
$$

In previous expressions AL (or CL) stands for the function which returns the sequence of attribute nodes (or the remainder of the insertion sequence), in its original order. PUL

represents a Pending Update List and *CCL* is a Constraint Check List. *CCL* has to be evaluated before evaluating PUL. *CC* is a function that is implemented by Constraints Checker Module in Section 4.3. The definition of CC function is denoted in Section 4.6.3.

The semantics of Insert Expression as stated in [12] is described below. The position of the inserted nodes is determined as follows [12]:

- If before (or after) is specified:
  The inserted nodes become the preceding (or following) siblings of the target node.
  - If multiple nodes are inserted by a single insert expression, the nodes remain adjacent and their order preserves the node ordering of the source expression.
  - If multiple groups of nodes are inserted by multiple insert expressions in the same snapshot, adjacency and ordering of nodes within each group is preserved but ordering among the groups is implementation-dependent.
- If as first into (or as last into) is specified:
  The inserted nodes become the first (or last) children of the target node.
  - If multiple nodes are inserted by a single insert expression, the nodes remain adjacent and their order preserves the node ordering of the source expression.
  - If multiple groups of nodes are inserted by multiple insert expressions in the same snapshot, adjacency and ordering of nodes within each group is preserved but ordering among the groups is implementation-dependent.
- If into is specified without as first or as last:
  The inserted nodes become children of the target node.
  - If multiple nodes are inserted by a single insert expression, their order preserves the node ordering of the source expression.

The positions of the inserted nodes are chosen so as not to interfere with the intended position of nodes that are inserted with the specification before, after, as first into, or as last into. For example, If node B is inserted "after node A", no other node will be inserted between nodes A and B unless it is also inserted "after node A".
Subject to the above constraints, the positions of the inserted nodes among the children of the target node are implementation-dependent.

Example 4.6.1. Insert Expression
Insert a year element after the publisher of the first book.

```
insert node <year>2005</year>
        after fn:doc("bib.xml")/books/book[1]/publisher
```

Navigating by means of several bound variables, insert a new police report into the list of police reports for a particular accident.

```
insert node $new-police-report
   as last into fn:doc("insurance.xml")/policies
      /policy[id = $pid]
      /driver[license = $license]
      /accident[date = $accdate]
      /police-reports
```

The semantics of an insert expression are as follows:

SourceExpr (SEN) must be a simple expression; otherwise a static error is raised [err:XUST0001] (for details see [12]). *SEN* is evaluated as though it were an enclosed expression in an element constructor. The result of this step is either an error or a sequence of nodes to be inserted, called the insertion sequence. If the insertion sequence contains a document node, the document node is replaced in the insertion sequence by its children. If the insertion sequence contains an attribute node following a node that is not an attribute node, a type error is raised [err:XUTY0004](for details see [12]). Let $alist be the sequence of attribute nodes in the insertion sequence. Let $clist be the remainder of the insertion sequence, in its original order.

Note 4.6.1. Either $alist or $clist or both may be empty.

The *Target Expression* (TEN) must be a simple expression; otherwise a static error is raised [err:XUST0001]. The target expression is evaluated and checked as follows:

- If the result is an empty sequence, [err:XUDY0027] is raised.
- If any form of into is specified, the result must be a single element or document node; any other non-empty result raises a type error [err:XUTY0005].

- If before or after is specified, the result must be a single element, text, comment, or processing instruction node; any other non-empty result raises a type error [err:XUTY0006].
- If before or after is specified, the node returned by the target expression must have a non-empty parent property [err:XUDY0029].

Let $target be the node returned by the *TEN*.
If $alist is not empty and any form of into is specified, the following checks are performed:

- $target must be an element node [err:XUTY0022].
- No attribute node in $alist may have a QName whose implied namespace binding conflicts with a namespace binding in the "namespaces" property of $target [err:XUDY0023], unless the namespace prefix for the attribute is absent.
- Multiple attribute nodes in $alist must not have QNames whose implied namespace bindings conflict with each other [err:XUDY0024].

If $alist is not empty and before or after is specified, the following checks are performed:

- parent($target) must be an element node [err:XUDY0030].
- No attribute node in $alist may have a QName whose implied namespace binding conflicts with a namespace binding in the "namespaces" property of parent($target) [err:XUDY0023] unless the namespace prefix for the attribute is absent.
- Multiple attribute nodes in $alist must not have QNames whose implied namespace bindings conflict with each other [err:XUDY0024].

The result of the insert expression is an empty XDM instance and a pending update list constructed as follows[3]:

- If *as first into* is specified, the pending update list consists of the following update primitives:
  - If $alist is not empty, upd:insertAttributes($target, $alist)

[3] The construction is described formally in the beginning of this section

  - If \$clist is not empty, upd:insertIntoAsFirst(\$target, \$clist)

- If *as last into* is specified, the pending update list consists of the following update primitives:
  - If \$alist is not empty, upd:insertAttributes(\$target, \$alist)
  - If \$clist is not empty, upd:insertIntoAsLast(\$target, \$clist)

- If *into* is specified with neither as first nor as last, the pending update list consists of the following update primitives:
  - If \$alist is not empty, upd:insertAttributes(\$target, \$alist)
  - If \$clist is not empty, upd:insertInto(\$target, \$clist)

- If *before* is specified, let \$parent be the parent node of \$target. The pending update list consists of the following update primitives:
  - If \$alist is not empty, upd:insertAttributes(\$parent, \$alist)
  - If \$clist is not empty, upd:insertBefore(\$target, \$clist)

- If *after* is specified, let \$parent be the parent node of \$target. The pending update list consists of the following update primitives:
  - If \$alist is not empty, upd:insertAttributes(\$parent, \$alist)
  - If \$clist is not empty, upd:insertAfter(\$target, \$clist)

4.6.1.2 Delete Expression

$$
\begin{aligned}
[\![delete\, node\, TE]\!]_{xquf}(g,l) = CC(g[PUL \leftarrow & \bigcup_{u \in [\![TE]\!](g,l) \,\wedge\, hasParent(u)} \{\text{u:del}(u)\}, \\
& CCL \leftarrow \{< delete, [\![TE]\!]_{xque}(g,l) >\}], l)
\end{aligned}
$$

Example 4.6.2. Delete Expression
Delete the last author of the first book in a given bibliography.

```
delete node fn:doc("bib.xml")/books/book[1]/author[last()]
```

Delete all email messages that are more than 365 days old.

```
delete nodes /email/message
      [fn:currentDate() - date > xs:dayTimeDuration("P365D")]
```

The semantics of a delete expression are as follows:

- The *Target Expression* (TE) must be a simple expression; otherwise a static error is raised [err:XUST0001]. The *TE* is evaluated. The result must be a sequence of zero or more nodes; otherwise a type error is raised [err:XUTY0007]. Let $tlist be the list of nodes returned by the *TE*.

- If any node in $tlist has no parent, it is removed from $tlist (and is thus ignored in the following step).

- A new pending update list is created. For each node $tnode in $tlist, the following update primitive is appended to the pending update list: upd:delete($tnode). The resulting pending update list (together with an empty XDM instance) is the result of the delete expression.

Note 4.6.2. Since node deletions do not become effective until the end of a snapshot, they have no effect on variable bindings or on the set of available documents or collections within the current query.

The semantics of a delete expression are defined in terms of their effect on an XDM instance: the deleted nodes are detached from their parents after completion of the current query.

4.6.1.3 Replace Expression

$$
\begin{aligned}
&[\![replace\, node\, TE\; with\; ES]\!]_{xquf}(g, l) = \\
&\qquad = CC(g[PUL \leftarrow \{\text{u:rN}([\![TE]\!]_{xque}(g, l), [\![ES]\!]_{xque}(g, l))\}, \\
&\qquad\qquad CCL \leftarrow \{< replace, [\![TE]\!]_{xque}(g, l), [\![ES]\!]_{xque}(g, l) >\}], l)
\end{aligned}
$$

$$[\![replace\ value\ of\ node\ TE\ with\ ES]\!]_{xquf}(g,l) =$$
$$= \frac{TypeOf([\![TE]\!]_{xque}(g,l)) = ElNode}{CC\Big(\begin{array}{l} g[PUL \leftarrow \{\text{u:rEC}([\![TE]\!]_{xque}(g,l), [\![ES]\!]_{xque}(g,l))\}, \\ \quad CCL \leftarrow \{< replaceV, [\![TE]\!]_{xque}(g,l), [\![ES]\!]_{xque}(g,l) >\}], l\end{array}\Big)}$$

$$[\![replace\ value\ of\ node\ TE\ with\ ES]\!]_{xquf}(g,l) =$$
$$= \frac{TypeOf([\![TE]\!]_{xque}(g,l)) \neq ElNode}{CC\Big(\begin{array}{l} g[PUL \leftarrow \{\text{u:rV}([\![TE]\!]_{xque}(g,l), [\![ES]\!]_{xque}(g,l))\}, \\ \quad CCL \leftarrow \{< replaceV, [\![TE]\!]_{xque}(g,l), [\![ES]\!]_{xque}(g,l) >\}], l\end{array}\Big)}$$

The semantics of Replace Expressions are as follows:

A replace expression is an updating expression. A replace expression has two forms, depending on whether *value of* clause is specified.

Replacing a Node
If *value of* is not specified, a replace expression replaces one node with a new sequence of zero or more nodes. The replacement nodes occupy the position in the node hierarchy that was formerly occupied by the node that was replaced. For this reason, an attribute node can be replaced only by zero or more attribute nodes, and an element, text, comment, or processing instruction node can be replaced only by zero or more element, text, comment, or processing instruction nodes.

Example 4.6.3. Replacing a Node
Replace the publisher of the first book with the publisher of the second book.

```
replace node fn:doc("bib.xml")/books/book[1]/publisher
with fn:doc("bib.xml")/books/book[2]/publisher
```

The semantics of this form of replace expression are as follows:

The expression following the keyword *with* must be a simple expression; otherwise a static error is raised [err:XUST0001]. This expression is evaluated as though it were an enclosed expression in an element constructor. Let $rlist be the node sequence that results from this evaluation. If $rlist contains a document node, the document node is replaced in $rlist by its children.

The target expression must be a simple expression; otherwise a static error is raised [err:XUST0001]. The target expression is evaluated and checked as follows:

- If the result is an empty sequence, [err:XUDY0027] is raised.
- If the result is non-empty and does not consist of a single element, attribute, text, comment, or processing instruction node, [err:XUTY0008] is raised.
- If the result consists of a node whose parent property is empty, [err:XUDY0009] is raised.

Let $target be the node returned by the target expression, and let $parent be its parent node.

- If $target is an element, text, comment, or processing instruction node, then $rlist must consist exclusively of zero or more element, text, comment, or processing instruction nodes [err:XUTY0010].
- If $target is an attribute node, then:
  - $rlist must consist exclusively of zero or more attribute nodes [err:XUTY0011].
  - No attribute node in $rlist may have a QName whose implied namespace binding conflicts with a namespace binding in the ”namespaces” property of $parent [err:XUDY0023] unless the namespace prefix for the attribute is absent.
  - Multiple attribute nodes in $rlist must not have QNames whose implied namespace bindings conflict with each other [err:XUDY0024].

The result of the replace expression is an empty XDM instance and a pending update list consisting of the following update primitive: upd:replaceNode($target, $rlist).

Replacing the Value of a Node

If *value of* is specified, a replace expression is used to modify the value of a node while preserving its node identity.

Example 4.6.4. Replacing the Value of a Node

Increase the price of the first book by ten percent.

```
replace value of node fn:doc("bib.xml")/books/book[1]/price
with fn:doc("bib.xml")/books/book[1]/price * 1.1
```

The semantics of this form of replace expression are as follows:

The expression following the keyword *with* must be a simple expression; otherwise a static error is raised [err:XUST0001]. This expression is evaluated as though it were the content expression of a text node constructor. The result of this step, in the absence of errors, is either a single text node or an empty sequence. Let $text be the result of this step.

The target expression must be a simple expression; otherwise a static error is raised [err:XUST0001]. The target expression is evaluated and checked as follows:

- If the result is an empty sequence, [err:XUDY0027] is raised.
- If the result is non-empty and does not consist of a single element, attribute, text, comment, or processing instruction node, [err:XUTY0008] is raised.

Let $target be the node returned by the target expression.

- If $target is an element node, the result of the replace expression is an empty XDM instance and a pending update list consisting of the following update primitive: upd:replaceElementContent($target, $text).
- If $target is an attribute, text, comment, or processing instruction node, let $string be the string value of the text node constructed in Step 1. If Step 1 did not construct a text node, let $string be a zero-length string. Then:
  - If $target is a comment node, and $string contains two adjacent hyphens or ends with a hyphen, a dynamic error is raised [err:XQDY0072].
  - If $target is a processing instruction node, and $string contains the substring "? >", a dynamic error is raised [err:XQDY0026].
  - In the absence of errors, the result of a replace expression is an empty XDM instance and a pending update list containing the following update primitive: upd:replaceValue($target, $string).

4.6.1.4 Rename Expression

$$\begin{aligned}
&[\![rename\, node\, TE\; as\; NNE]\!]_{xquf}(g,l) = \\
&\quad = CC(g[PUL \leftarrow \{u : ren([\![TE]\!]_{xque}(g,l), [\![NNE]\!]_{xque}(g,l))\}, \\
&\qquad\qquad CCL \leftarrow \{< rename, [\![TE]\!]_{xque}(g,l), [\![NNE]\!]_{xque}(g,l) >\}], l)
\end{aligned}$$

A rename expression replaces the name property of a data model node with a new QName. A rename expression is an updating expression.

Example 4.6.5. Rename the first author element of the first book to principal-author.

```
rename node fn:doc("bib.xml")/books/book[1]/author[1]
as "principal-author"
```

Example 4.6.6. Rename the first author element of the first book to the QName that is the value of the variable $newname.

```
rename node fn:doc("bib.xml")/books/book[1]/author[1]
as $newname
```

The target expression must be a simple expression; otherwise a static error is raised [err:XUST0001]. The target expression is evaluated and checked as follows:

- If the result is an empty sequence, [err:XUDY0027] is raised.
- If the result is non-empty and does not consist of a single element, attribute, or processing instruction node, [err:XUTY0012] is raised.

The semantics of rename expression is as follows:
Let $target be the node returned by the *Target Expression* (TE).

New Name Expression (NNE) must be a simple expression; otherwise a static error is raised [err:XUST0001]. *NNE* is processed as follows:

- If $target is an element node, let $QName be the result of evaluating *NNE* as though it were the name expression of a computed element constructor. If the namespace binding of $QName conflicts with any namespace binding in the namespaces property of $target, a dynamic error is raised [err:XUDY0023].

- If $target is an attribute node, let $QName be the result of evaluating *NNE* as though it were the name expression of a computed attribute constructor. If $QName has a non-absent namespace URI, and if the namespace binding of $QName conflicts with any namespace binding in the namespaces property of the parent (if any) of $target, a dynamic error is raised [err:XUDY0023].

- If $target is a processing instruction node, let $NCName be the result of evaluating *NNE* as though it were the name expression of a computed processing instruction constructor, and let $QName be defined as fn:QName((), $NCName).

The result of the rename expression is an empty XDM instance and a pending update list containing the following update primitive: upd:rename($target, $QName).

4.6.2 Update Operations' Semantics

In previous section we introduced a formal semantics of XQUF expressions. These expressions are executed in three steps. During the first step XQUF expressions are transformed to the list of update operations called Pending Update List (PUL) and the list of pairs of input expressions and operations called Constraints Check List(CCL). In the second step CCL is processed by Constraints Checker described in Section 4.6.3. If CCL evaluates without errors PUL executor is executed [4](third step) and the XDM instance is modified, otherwise error is thrown (*throw* function) and the execution is stopped. The Execution Flow of XQuery Update Facility is shown in Figure 4.3.

Update operations are elementary for the correct processing of updating expressions. They are used in the semantics definitions of XQUF expressions, but they are not directly available to users. XQuery Update Facility 1.0 specification provides semantics of these operations from the single user/transaction point of view. We extend their semantics by transaction processing. We assume Light-Weight XDM model described in Section 4.3, but its semantics specification can be easily extended for all objects of XDM. We also provide the semantics of operations in original meaning of W3C specification [12]. The semantics of the update operations are introduced. Transaction processing extensions are marked explicitly. During the second phase all operations in PUL are evaluated and the XDM instance is modified.

[4]PUL executor implements function upd:applyUpdates

Update operations consist of update primitives, which are the components of pending update lists, and update routines, which are used in defining XQuery semantics but do not appear on pending update lists [12].

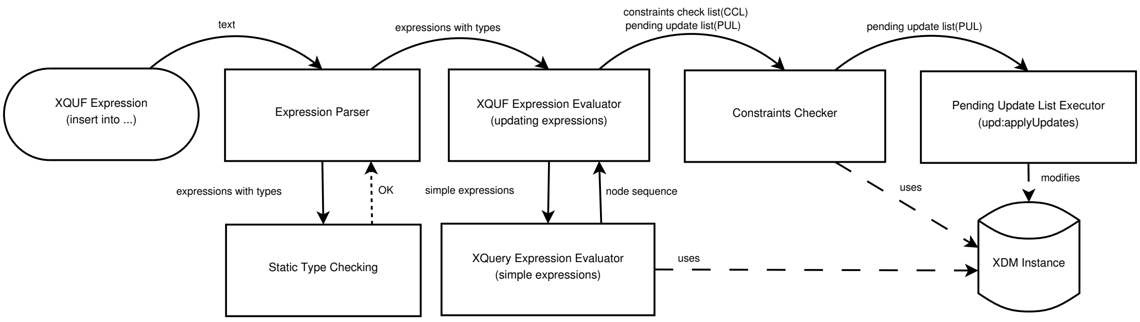


Figure 4.3: XQuery Update Facility Execution Flow, $\rightarrow$ data path, $- \rightarrow$ uses or modifies, $\dashrightarrow$ signal path

4.6.3 Constraints Checker

Constraints Checker is a module that implements CC function. This function checks constraints given for XQUF expressions [5] in the W3C specification [12]. The state s in semantics of CC is used to store internal state of Constraints Checker (e.g. constraints violation).

The semantics of CC function is following[6]:

$$
\begin{aligned}
CC(g[CCL_{/l[TRANS]} ==< op : REST >], l) &= \\
&= CC(OP(op, g[CCL_{/l[TRANS]} := REST], l)) \\
CC(g[CCL_{/l[TRANS]} == \emptyset], l[ERR == \emptyset]) &= \\
&= \text{upd:applyUpdates}(g[PUL_{/l[TRANS]}], ``strict", false, g, l) \\
CC(g[CCL_{/l[TRANS]} == \emptyset], l[ERR \neq \emptyset]) &= throw(g, l) \\
throw(g, l) &= printErr(g, l)
\end{aligned}
$$

[5] By expressions we mean source and target expressions.

[6] $CCL_{/l[TRANS]}$ means that the operations not belonging to $l[TRANS]$ are sieved out.

$$op \in \bigcup_{\forall Const} < Const, [\![E_1]\!](g,l), [\![E_2]\!](g,l) >$$

$$\begin{aligned} Const = &\{insertFI, insertLI, insertI, insertB, insertA, delete, replace,\\ &replaceV, rename\}\end{aligned}$$

The *printErr* function prints the contents of the l[ERR] variable and terminates the evaluation of the expression.

4.6.3.1 Insert Expression Constraints Check

We use simple regular expressions [38] to identify constants in this section. For example, the expression *insert** covers all constants begining *insert*. So, all constants, such as insertA, insertB, insertI, insertLI and insertFI are matched by this expression. In some of the following expressions the conditional notation is not used according to readability and the paper width. We use a natural language instead of the notation. The Constraints Checker's semantics is adopted from [12].

SourceExpression (SEN) constraint check:

$$\frac{TypeOf(SE) \neq SimpleExpr}{OP(< insert*, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUST0001\}])}$$

- If the result of $[\![SE]\!]_{xque}(g,l)$ contains an attribute node following a node that is not an attribute node, an error is raised [12]:

$$\begin{aligned} OP(< insert*, [\![TE]\!]_{xque}(g,l), &[\![SE]\!]_{xque}(g,l) >, g, l) =\\ &= (g, l[ERR \leftarrow \{XUTY0004\}])\end{aligned}$$

TargetExpression (TEN) constraint check:

$$\frac{TypeOf(TE) \neq SimpleExpr}{OP(< insert*, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUST0001\}])}$$

$$\frac{[\![TE]\!]_{xque}(g,l) == ()}{OP(< insert*, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUDY0027\}])}$$

$$\frac{[\![TE]\!]_{xque}(g,l) == (g, l[RES \neq (x : (Element|Document))])}{OP(< insert*I, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUTY0005\}])}$$

$$\frac{[\![TE]\!]_{xque}(g,l) == (g, l[RES \neq (x : (Element|Text|Comment|PI))])}{OP(< insert(B|A), [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUTY0006\}])}$$

$$\frac{([\![TE]\!]_{xque}(g,l) == (g, l[RES == (node)])) \wedge parent(node) == \epsilon}{OP(< insert(B|A), [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUDY0029\}])}$$

If $AL([\![SE]\!]_{xque}(g,l))$ [7] is not empty, the folowing checks are performed:

$$\frac{[\![TE]\!]_{xque}(g,l) == (g, l[RES \neq (x : Element)])}{OP(< insert*I, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUTY0022\}])}$$

- No attribute node in $AL([\![SE]\!]_{xque}(g,l))$ may have a QName whose implied namespace binding conflicts with a namespace binding in the "namespaces" property of $[\![TE]\!]_{xque}(g,l)$, otherwise an error is raised, unless the namespace prefix for the attribute is absent [12]:

$$OP(< insert*I, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = \\ = (g, l[ERR \leftarrow \{XUDY0023\}])$$

- Multiple attribute nodes in $AL([\![SE]\!]_{xque}(g,l)$ must not have QNames whose implied namespace bindings conflict with each other, otherwise an error is raised [12]:

$$OP(< insert*, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = \\ = (g, l[ERR \leftarrow \{XUDY0024\}])$$

[7] Function AL returns sequence of attributes.

$$\frac{([\![TE]\!]_{xque}(g,l) == (g,l[RES == (node)])) \wedge TypeOf(parent(node)) \neq Element}{OP(< insert{*}I, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUDY0030\}])}$$

- No attribute node in $AL([\![SE]\!]_{xque}(g,l))$ may have a QName whose implied namespace binding conflicts with a namespace binding in the "namespaces" property of $parent([\![TE]\!]_{xque}(g,l))$, otherwise an error is raised, unless the namespace prefix for the attribute is absent [12]:

$$OP(< insert{*}I, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) =$$
$$= (g, l[ERR \leftarrow \{XUDY0023\}])$$

4.6.3.2 Delete Expression Constraints Check

$$\frac{TypeOf(TE) \neq SimpleExpr}{OP(< delete, [\![TE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUST0001\}])}$$

- The result must be a sequence of zero or more nodes, otherwise an error is raised [12]:

$$\frac{[\![TE]\!]_{xque}(g,l) == (g,l[RES \neq (x*)])}{OP(< delete, [\![TE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUTY0007\}])}$$

4.6.3.3 Replace Expression Constraints Check

$$\frac{TypeOf(SE) \neq SimpleExpr}{OP(< replace{*}, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUST0001\}])}$$

$$\frac{TypeOf(TE) \neq SimpleExpr}{OP(< replace{*}, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUST0001\}])}$$

$$\frac{[\![TE]\!]_{xque}(g,l) == (g,l[RES == ()])}{OP(< replace{*}, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUDY0027\}])}$$

$$\frac{[\![TE]\!]_{xque}(g,l) == (g, l[RES == (node : (Element|Text|Comment|PI))])}{OP(< replace*, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUTY0008\}])}$$

$$\frac{([\![TE]\!]_{xque}(g,l) == (g, l[RES == (node)])) \wedge parent(node) == \epsilon}{OP(< replace*, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = (g, l[ERR \leftarrow \{XUDY0009\}])}$$

- If the result of $[\![TE]\!]_{xque}(g,l)$ is an element, text, comment, or processing instruction node, then the result of the $[\![SE]\!]_{xque}(g,l)$ must consist exclusively of zero or more element, text, comment, or processing instruction nodes, otherwise an error is raised [12]:

$$\begin{aligned} &OP(< replace*, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = \\ &\qquad = (g, l[ERR \leftarrow \{XUTY0010\}]) \end{aligned}$$

- If the result of $[\![TE]\!]_{xque}(g,l)$ is an attribute node, then the result of the $[\![SE]\!]_{xque}(g,l)$ must consist exclusively of zero or more attribute nodes, otherwise an error is raised [12]:

$$\begin{aligned} &OP(< replace*, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = \\ &\qquad = (g, l[ERR \leftarrow \{XUTY0011\}]) \end{aligned}$$

- If the result of $[\![TE]\!]_{xque}(g,l)$ is an attribute node, then no attribute node in the result of the $[\![SE]\!]_{xque}(g,l)$ may have a QName whose implied namespace binding conflicts with a namespace binding in the "namespaces" property of $parent([\![TE]\!]_{xque}(g,l))$ unless the namespace prefix for the attribute is absent, otherwise an error is raised [12]:

$$\begin{aligned} &OP(< replace*, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) = \\ &\qquad = (g, l[ERR \leftarrow \{XUDY0023\}]) \end{aligned}$$

- If the result of $[\![TE]\!]_{xque}$ is an attribute node, then multiple attribute nodes in $[\![SE]\!]_{xque}$ must not have QNames whose implied namespace bindings conflict with each other, otherwise an error is raised [12]:

$$OP(< replace*, [\![TE]\!]_{xque}(g,l), [\![SE]\!]_{xque}(g,l) >, g, l) =$$
$$= (g, l[ERR \leftarrow \{XUDY0024\}])$$

4.6.4 Update Primitives' Semantics

The update primitives described in this section are held on pending update lists. When an update primitive is held on a pending update list, its node operands are represented by their node identities [12]. The semantics of an update primitive do not become effective until their pending update list is processed by the *upd:applyUpdates* routine. We define transaction semantics for each update primitive. We introduce a function $Lock(node, lock\_mode)$ which implements XQUF-LP locking protocol as specified in Section 4.7.3.

upd:insertBefore

`upd:insertBefore($target as node(), $content as node()+)`

Summary

Inserts $content immediately before $target.

Constraints

$target must be an element, text, processing instruction, or comment node with a non-empty parent property. $content must be a sequence containing only element, text, processing instruction, and comment nodes.

Semantics

Effects on nodes in $content:

- For each node in $content, the parent property is set to parent($target).
- If the type-name property of parent($target) is xs:untyped, then upd:setToUntyped() is invoked on each element or attribute node in $content.

Effects on parent($target):

- The children property of parent($target) is modified to add the nodes in $content just before $target, preserving their order.
- If at least one of the nodes in $content is an element or text node, upd:removeType(parent($target)) is invoked.
- All the namespace bindings of parent($target) are marked for namespace propagation.

Transaction Semantics

$$\mathit{u{:}iB}(\$target, \$content, g, l) = \mathit{Lock}(\$content, X, \mathit{Lock}(\$target, SB, g, l))$$

upd:insertAfter

```
upd:insertAfter($target as node(), $content as node()+)
```

Summary

Inserts $content immediately after $target.

Constraints

$target must be an element, text, processing instruction, or comment node with a non-empty parent property. $content must be a sequence containing only element, text, processing instruction, and comment nodes.

Semantics

The semantics of upd:insertAfter are identical to the semantics of upd:insertBefore, except that Rule 2a (Effects on parent) is changed as follows:

The children property of parent($target) is modified to add the nodes in $content just after $target, preserving their order.

Transaction Semantics

$$u{:}iA(\$target, \$content, g, l) = Lock(\$content, X, Lock(\$target, SA, g, l))$$

upd:insertInto

```
upd:insertInto(
   $target as node(),
   $content as node()+)
```

Summary

Inserts $content as the children of $target, in an implementation-dependent position.

Constraints

$target must be an element or document node. $content must be a sequence containing only element, text, processing instruction, and comment nodes.

Semantics

The semantics of upd:insertInto are identical to the semantics of upd:insertBefore, except that $target is substituted everywhere for parent($target), and Rule 2a (Effects on parent) is changed as follows:

The children property of $target is changed to add the nodes in $content in implementation-dependent positions, preserving their relative order.

Transaction Semantics

$$\textit{u:iI}(\$target, \$content, g, l) = \textit{Lock}(\$content, X, \textit{Lock}(\$target, SI, g, l))$$

upd:insertIntoAsFirst

```
upd:insertIntoAsFirst(
   $target as node(),
   $content as node()+)
```

Summary

Inserts \$content as the first children of \$target.

Constraints

\$target must be an element or document node. \$content must be a sequence containing only element, text, processing instruction, and comment nodes.

Semantics

The semantics of upd:insertIntoAsFirst are identical to the semantics of upd:insertBefore, except that \$target is substituted everywhere for parent(\$target), and Rule 2a (Effects on parent) is changed as follows:

The children property of \$target is changed to add the nodes in \$content as the first children, preserving their order.

Transaction Semantics

$$\mathit{u{:}iIAF}(\$target, \$content, g, l) = \mathit{Lock}(\$content, X, \mathit{Lock}(\$target, SIF, g, l))$$

upd:insertIntoAsLast

```
upd:insertIntoAsLast(
       $target as node(),
       $content as node()+)
```

Summary

Inserts $content as the last children of $target.

Constraints

$target must be an element or document node. $content must be a sequence containing only element, text, processing instruction, and comment nodes.

Semantics

The semantics of upd:insertIntoAsLast are identical to the semantics of upd:insertBefore, except that $target is substituted everywhere for parent($target), and Rule 2a (Effects on parent) is changed as follows:

The children property of $target is changed to add the nodes in $content as the last children, preserving their order.

Transaction Semantics

$$\textit{u:iIAL}(\$target, \$content, g, l) = Lock(\$content, X, Lock(\$target, SIL, g, l))$$

upd:insertAttributes

```
upd:insertAttributes(
   $target as element(),
       $content as attribute()+)
```

Summary

Inserts $content as attributes of $target.

Constraints

None

Semantics

- For each node $A in $content:

  The parent property of $A is set to $target.

  If the type-name property of $target is xs:untyped, then upd:setToUntyped($A) is invoked.

- The following properties of $target are changed:

  attributes: Modified to add the nodes in $content.

  namespaces: Modified to add namespace bindings for any attribute namespace prefixes in $content that did not already have bindings. These bindings are marked for namespace propagation.

  upd:removeType($target) is invoked.

Transaction Semantics

$$\mathit{u{:}iAttrs}(\$target, \$content, g, l) = \mathit{Lock}(\$content, X, \mathit{Lock}(\$target, SIT, g, l))$$

upd:delete

```
upd:delete(
   $target as node())
```

Constraints

None

Semantics

- If $target has a parent node $P, then:

  The parent property of $target is set to empty.

  If $target is an attribute node, the attributes property of $P is modified to remove $target.

  If $target is a non-attribute node, the children property of $P is modified to remove $target.

  If $target is an element, attribute, or text node, and $P is an element node, then upd:removeType($P) is invoked.

- If $target has no parent, the XDM instance is unchanged.

Note 4.6.3. Deleted nodes are detached from their parent nodes; however, a node deletion has no effect on variable bindings or on the set of available documents or collections during processing of the current query.

Note 4.6.4. Multiple upd:delete operations may be applied to the same node during execution of a query; this is not an error.

Transaction Semantics

$$\mathit{u{:}del}(\$target, g, l) = \mathit{Lock}(\$target, XT, g, l)$$

upd:replaceNode

```
upd:replaceNode(
   $target as node(),
   $replacement as node()*)
```

Summary

Replaces $target with $replacement.

Constraints

$target must be a node that has a parent. If $target is an attribute node, $replacement must consist of zero or more attribute nodes. If $target is an element, text, comment, or processing instruction node, $replacement must consist of zero or more element, text, comment, or processing instruction nodes.

Semantics

- Effects on nodes in $replacement:

  For each node in $replacement, the parent property is set to parent($target).

  If the type-name property of parent($target) is xs:untyped, then upd:setToUntyped() is invoked on each node in $replacement.

- Effect on $target:

  The parent property of $target is set to empty.

- Effects on parent($target):

  If $target is an attribute node, the attributes property of parent($target) is modified by removing $target and adding the nodes in $replacement (if any).

  If $target is an attribute node, the namespaces property of parent($target) is modified to add namespace bindings for any attribute namespace prefixes in $replacement that did not already have bindings. These bindings are marked for namespace propagation.

  If $target is an element, text, comment, or processing instruction node, the children property of parent($target) is modified by removing $target and adding the nodes in $replacement (if any) in the former position of $target, preserving their order.

If $target or any node in $replacement is an element, attribute, or text node, upd:removeType(parent($target)) is invoked.

Transaction Semantics

$$\mathit{u{:}rN}(\$target, \$content, g, l) = \mathit{Lock}(\$target, XT, g, l)$$

upd:replaceValue

```
upd:replaceValue(
   $target as node(),
   $string-value as xs:string)
```

Summary

Replaces the string value of $target with $string-value.

Constraints

$target must be an attribute, text, comment, or processing instruction node.

Semantics

- If $target is an attribute node:

  string-value of $target is set to $string-value.

  upd:removeType($target) is invoked.

- If $target is a text, comment, or processing instruction node: content of $target is set to $string-value.

- If $target is a text node that has a parent, upd:removeType(parent($target)) is invoked.

Transaction Semantics

$$\mathit{u{:}rV}(\$target, \$content, g, l) = \mathit{Lock}(\$target, X, g, l)$$

upd:replaceElementContent

```
upd:replaceElementContent(
   $target as element(),
   $text as text()?)
```

Summary

Replaces the existing children of the element node $target by the optional text node $text. The attributes of $target are not affected.

Constraints

None.

Semantics

- For each node $C that is a child of $target, the parent property of $C is set to empty.
- The parent property of $text is set to $target.
- Effects on $target:

  children is set to consist exclusively of $text. If $text is an empty sequence, then $target has no children.

  typed-value and string-value are set to the content property of $text. If $text is an empty sequence, then typed-value is an empty sequence and string-value is an empty string.

  upd:removeType($target) is invoked.

Transaction Semantics

$$\textit{u:rEC}(\$target, \$content, g, l) = \textit{Lock}(\$target, X, g, l)$$

upd:rename

```
upd:rename(
   $target as node(),
   $newName as xs:QName)
```

Summary

Changes the node-name of $target to $newName.

Constraints

$target must be an element, attribute, or processing instruction node.

Semantics

- If $target is an element node:

  node-name of $target is set to $newName.

  upd:removeType($target) is invoked.

  If $newname has no prefix and no namespace URI, the namespaces property of $target is modified by removing the binding (if any) for the empty prefix.

  The namespaces property of $target is modified to add a namespace binding derived from $newName, if this binding did not already exist. This binding is marked for namespace propagation.

- If $target is an attribute node:

  node-name of $target is set to $newName.

  upd:removeType($target) is invoked.

  If $newName is xml:id, the is-id property of $target is set to true.

  If $target has a parent, the namespaces property of parent($target) is modified to add a namespace binding derived from $newName, if this binding did not already exist. This binding is marked for namespace propagation.

- If $target is a processing instruction node, its target property is set to the local part of $newName.

Note 4.6.5. At the end of a snapshot, if multiple attribute nodes with the same parent have the same qualified name, an error will be raised by upd:applyUpdates.

Transaction Semantics

$$u{:}ren(\$target, \$content, g, l) = Lock(\$target, X, g, l)$$

upd:put

```
upd:put(
   $node as node(),
   $uri as xs:string)
```

Summary

The XDM node tree rooted at $node is stored to the location specified by $uri.

Constraints

$uri must be a valid absolute URI.

Semantics

The external effects of upd:put are implementation-defined, since they occur outside the domain of XQuery. The intent is that, if upd:put is invoked on a document node and no error is raised, a subsequent query can access the stored document by invoking fn:doc with the same URI.

Transaction Semantics

Transaction semantics is not defined for this operation, because the effects occur outside the domain of the database and XQuery. The correct transaction processing has to be provided by the server where $uri targets.

4.6.5 Update Routines Semantics

The update routines are helper functions used for processing of pending update list(s). They cannot appear in pending update list. Their importance lies in processing XDM manipulation operations described by update primitives in previous section.

upd:mergeUpdates

```
upd:mergeUpdates(
   $pul1 as pending-update-list,
   $pul2 as pending-update-list)
```

Summary

Merges two pending update lists. The routine is invocated to merge all pending update lists generated by successive calls of a return expression in a FLWOR expression.

Constraints

None.

Semantics

- The two pending update lists are merged and a single pending update list containing all the update primitives from both lists is returned.

- Optionally, *upd:mergeUpdates* may raise a dynamic error if any of the following conditions are detected:

  Two or more *upd:rename* primitives on the merged list have the same target node [err:XUDY0015].

  Two or more *upd:replaceNode* primitives on the merged list have the same target node [err:XUDY0016].

  Two or more *upd:replaceValue* primitives on the merged list have the same target node [err:XUDY0017].

  Two or more *upd:replaceElementContent* primitives on the merged list have the same target node [err:XUDY0017].

  Two or more *upd:put* primitives on the merged list have the same $uri operand [err:XUDY0031].

Two or more primitives on the merged list create conflicting namespace bindings for the same element node [err:XUDY0024]. The following kinds of primitives create namespace bindings:

upd:insertAttributes creates one namespace binding on the $target element corresponding to the implied namespace binding of the name of each attribute node in $content if the name has a non-empty prefix.

upd:replaceNode creates one namespace binding on the parent($target) element corresponding to the implied namespace binding of the name of each attribute node in $replacement if the name has a non-empty prefix.

upd:rename creates a namespace binding on $target, or on the parent (if any) of $target if $target is an attribute node, corresponding to the implied namespace binding of $newName. However, if $target is an attribute and its name has an empty prefix, the namespace binding is not created.

Transaction Semantics

This update routine does not have a transaction semantics because it does not cause any update effects to the XDM instance. It only merges unprocessed pending update lists following the rules specified above.

upd:applyUpdates

```
upd:applyUpdates(
   $pul as pending-update-list,
   $revalidation-mode as xs:string,
   $inherit-namespaces as xs:boolean)
```

Summary

This routine ends a snapshot by making effective the semantics of all the update primitives on a pending update list and by revalidating the resulting XDM instance.

Constraints

$revalidation-mode must be "strict", "lax", or "skip"

Semantics

- Checks the update primitives on $pul for compatibility. Raises a dynamic error if any of the following conditions are detected:

  Two or more *upd:rename* primitives on $pul have the same target node [err:XUDY0015].

  Two or more *upd:replaceNode* primitives on $pul have the same target node [err:XUDY0016].

  Two or more *upd:replaceValue* primitives on $pul have the same target node [err:XUDY0017].

  Two or more *upd:replaceElementContent* primitives on $pul have the same target node [err:XUDY0017].

  Two or more *upd:put* primitives on the merged list have the same $uri operand [err:XUDY0031].

  Two or more primitives on $pul create conflicting namespace bindings for the same element node [err:XUDY0024]. The following kinds of primitives create namespace bindings:

  upd:insertAttributes creates one namespace binding on the parent($target) element corresponding to the implied namespace binding of the name of each attribute node in $content if the name has a non-empty prefix.

upd:replaceNode creates one namespace binding on the $target element corresponding to the implied namespace binding of the name of each attribute node in $replacement if the name has a non-empty prefix.

upd:rename creates a namespace binding on $target, or on the parent (if any) of $target if $target is an attribute node, corresponding to the implied namespace binding of $newName. However, if $target is an attribute and its name has an empty prefix, the namespace binding is not created.

- The semantics of all update primitives on $pul, other than *upd:put* primitives, are made effective in the following order:

  First, all *upd:insertInto*, *upd:insertAttributes*, *upd:replaceValue*, and *upd:rename* primitives are applied.

  Next, all *upd:insertBefore*, *upd:insertAfter*, *upd:insertIntoAsFirst*, and *upd:insertIntoAsLast* primitives are applied.

  Next, all *upd:replaceNode* primitives are applied.

  Next, all *upd:replaceElementContent* primitives are applied.

  Next, all *upd:delete* primitives are applied.

- If, as a net result of the above steps, the children property of some node contains adjacent text nodes, these adjacent text nodes are merged into a single text node. The string-value of the resulting text node is the concatenated string-values of the adjacent text nodes, with no intervening space added. The node identity of the resulting text node is implementation-dependent.

- If, as a net result of the above steps, the children property of some node contains an empty text node, that empty text node is deleted from the children property.

- If, after applying the updates, any XDM instance (including a node that has been deleted or detached from its parent, or that is a descendant of such a node) violates any constraint specified in [21] , a dynamic error is raised [err:XUDY0021].

  Note 4.6.6. For example, a data model constraint violation might occur if multiple attributes with the same parent have the same qualified name.

  Note 4.6.7. During processing of a pending update list, an XDM instance may temporarily violate a data model constraint. An error is raised only if a constraint remains unsatisfied after all update primitives other than upd:put have been applied.

- If \$inherit-namespaces is true, then *upd:propagateNamespace(\$element, \$prefix, \$uri)* is invoked for each namespace binding that was marked for namespace propagation, except for namespace bindings associated with the empty prefix, where \$element is the element node on which the namespace binding appears, \$prefix is the namespace prefix, and \$uri is the namespace URI. Each of these nodes is then unmarked.

- For each document or element node \$top that was marked for revalidation by one of the earlier steps, upd:revalidate(\$top, \$revalidation-mode) is invoked. Each of these nodes is then unmarked.

- As the final step, all upd:put primitives on \$pul are applied.

- The *upd:applyUpdates* operation is atomic with respect to the data model. In other words, if *upd:applyUpdates* terminates normally, the resulting XDM instance reflects the result of all update primitives; but if upd:applyUpdates raises an error, the resulting XDM instance reflects no changes. Atomicity is guaranteed only with respect to operations on XDM instances, and only with respect to error conditions specified in this document.

  Note 4.6.8. The results of implementation-dependent error conditions such as exceeding resource limits are beyond the scope of this specification.

- Propagation of XDM changes to an underlying persistent store is beyond the scope of this specification. For example, the effect on persistent storage of deleting a node that has no parent is beyond the scope of this specification.

Transaction semantics

Transaction semantics for *upd:applyUpdates* routine is divided into two parts. First, $CheckPUL$ function is evaluated. This function implements standard *upd:applyUpdates* semantics as described in the previous paragraph. If $CheckPUL$ function detects errors in *Pending Update List*, then they are propagated in context variable *ERR* and processing is interrupted. If no errors are detected then modified *Pending Update List* is returned and *upd:appUpT* function simply iterates through modified *Pending Update List* and evaluates atomic update operations.

$$OP : \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

$$
\begin{aligned}
&\text{upd:applyUpdates}(PUL, RM, IN, g, l) = \\
&\qquad = \text{upd:appUpT}(CheckPUL(PUL, RM, IN, g, l)) \\
&\quad \text{upd:appUpT}(g[PUL == < OP : REST >], l) = \\
&\qquad = \text{upd:appUpT}(OP(g[PUL := REST], l)) \\
&\quad \text{upd:appUpT}(g[PUL == \emptyset], l) = (g, l)
\end{aligned}
$$

4.7 XQuery Update Facility Transaction Extension

In this section we provide a draft of XQuery Transaction Control Language (XTCL) which could be a part of XQUF in the future.

Motivation XQuery Update Facility (XQUF) is defined as a language extension of XQuery for updates. In XQUF is a lack of language constructions for the transaction processing. We think that for better usability and implementability of XQUF across platforms these transaction statements have to be defined in the language specification. In current XQUF 1.0 [12] is said that the transaction processing is out of scope the language specification and is up to the implementor. We assume that the basic transaction language specification should be provided by the language itself or as an optional extension. Our assumption is based on a good practice from relational databases where SQL-TCL (Transaction Control Language) is a part of SQL language specification [52].

Implementation To implement XTCL, EBNF grammar of XQUF has to be extended by appropriate grammar rules. XTCL supports basic transaction commands BEGIN, COMMIT and ROLLBACK and prologue command SET TRANSACTION ISOLATION LEVEL. In the next step we define appropriate semantics for these commands in the context of XQUF semantics as introduced in the previous section.

4.7.1 XQuery Transaction Control Language - Grammar

BEGIN Starts a new transaction. Transactions cannot be nested. If omitted a new transaction is started.

COMMIT Commit ends a current transaction and makes all changes visible to other users. It can be omitted.

ROLLBACK Rollback ends a current transaction and undoes all changes made by this transaction.

XQUF grammar is modified the following way:

```
[31]    Expr  ::=  ExprSingle ("," ExprSingle)*
```

```
[32]    ExprSingle  ::= FLWORExpr
                       | QuantifiedExpr
                       | TypeswitchExpr
                       | IfExpr
                       | InsertExpr
                       | DeleteExpr
                       | RenameExpr
                       | ReplaceExpr
                       | TransformExpr
                       | OrExpr
                       | BeginExpr
                       | CommitExpr
                       | RollbackExpr
[200]   BeginExpr ::= "BEGIN"
[201]   CommitExpr ::= "COMMIT"
[202]   RollbackExpr ::= "ROLLBACK"
```

Three new expressions were added to the grammar: BeginExpr, CommitExpr and RollbackExpr. Its grammar is simple, they contain only a terminal keyword.

Expression SET TRANSACTION ISOLATION LEVEL is a prolog-related expression which can be evaluated only once in the beginning of a transaction.

Corresponding XQUF grammar is modified as:

```
[6]     Prolog  ::=  ((DefaultNamespaceDecl
                       | Setter
                       | NamespaceDecl
                       | Import) Separator)*
                     ((VarDecl | FunctionDecl | OptionDecl) Separator)*
[7]     Setter  ::=  BoundarySpaceDecl
                     | DefaultCollationDecl
                     | BaseURIDecl
                     | ConstructionDecl
                     | OrderingModeDecl
                     | EmptyOrderDecl
```

```
                     | RevalidationDecl
                     | CopyNamespacesDecl
                     | TransactionDecl
[203]   TransactionDecl ::= "declare" "transaction" "isolation" "level"
                          (("read" ("uncommitted" | "committed"))
                          |("repeatable" "read")
                          |("serializable")
```

A new setter expression was added into the grammar called TransactionDecl. It is part of Prolog rule. We define four standard isolation levels - read committed, read uncommitted, repeatable read and serializable.

4.7.2 XQuery Transaction Control Language - Semantics

Semantics of transaction control language expressions can be defined as follows:

$$beginTransaction : \mathcal{G}_{cont} \times \mathcal{L}_{cont} \times Transaction \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

$$beginTransaction(g, l, t) = (g[TRANS \leftarrow t], l[TRANS := t])$$

$$commitTransaction : \mathcal{G}_{cont} \times \mathcal{L}_{cont} \times Transaction \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

$$\begin{aligned} &commitTransaction(g, l[TRANS == t]) = \\ &\quad (g[WFG := removeTransaction(WFG, t), \\ &\quad TRANS := TRANS \setminus \{t\}, \\ &\quad LOCKS := LOCKS \setminus \{< el, lock, t >\}], l[TRANS := \epsilon, ERR := \emptyset]) \end{aligned}$$

$$abortTransaction : \mathcal{G}_{cont} \times \mathcal{L}_{cont} \times Transaction \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

$$
\begin{aligned}
&abortTransaction(g, l[TRANS == t]) = \\
&\qquad (g[WFG := removeTransaction(WFG, t), \\
&\qquad TRANS := TRANS \setminus \{t\}, \\
&\qquad LOCKS := LOCKS \setminus \{< el, lock, t >\}, \\
&\qquad PUL := PUL \setminus view(PUL, t)], l[TRANS := \epsilon, ERR := \emptyset])
\end{aligned}
$$

$$
\begin{aligned}
[\![BEGIN]\!](g, l == \epsilon) &= beginTransaction(g, l, createTrans(beginTrans())) \\
[\![COMMIT]\!](g, l) &= commitTransaction(\text{upd:applyUpdates}(g, l)) \\
[\![ROLLBACK]\!](g, l) &= abortTransaction(g, l)
\end{aligned}
$$

The helper function $beginTrans()$ generates a new transaction id. This unique transaction identifier is stored in the state variable $TRANS$. The symbole ϵ is used as undefined value for a variable. The function $view(PUL, TRANS)$ returns an ordered set of operations from PUL which were created by a transaction $TRANS$. ROLLBACK expression removes the transaction's operations from Pending Update List.

Example 4.7.1. Query with XTCL expression.

We suppose the following query:

```
let $q := /inventory/item[serialno = "123456"]/quantity
return
   ( replace value of node $q with ( ),
     insert node attribute xsi:nil {"true"} into $q )
```

by the semantics it is equivalent to:

```
BEGIN, let $q := /inventory/item[serialno = "123456"]/quantity
return
   ( replace value of node $q with ( ),
     insert node attribute xsi:nil {"true"} into $q ),
COMMIT
```

ROLLBACK expression can be used to prevent update if something goes wrong:

```
let $e:=/inventory/item[serialno = "123456"]
return(
if ($e/@last-updated < '2012-03-03')
then replace value of node
        \$e/last-updated with fn:currentDate()
else ROLLBACK)
```

4.7.3 Lock Function Semantics

Previous sections introduced usage of the *Lock* function. This function is the key to transaction processing in our specification. In real-world transactional system it should be implemented by the Lock Manager module. Its specification is given by three basic elements - Compatibility Matrix, Conversion Matrix and Locking Semantics. We adopted a granular lock protocol presented by Gray and Reuter in [23], taDOM Lock Protocol Protocol published by Haustein and Härder in [26], XLP protocol from Jea and Chen published in [36] and Sedna Lock Protocol introduced by Pleshachkov and Novak in [48]. The compatibility matrix of this new lock protocol is shown in Figure 4.1. The symbols + and - in the matrix cells means that granted and requested lock modes are compatible or incompatible respectively. The symbol x mean the conditional compatibility specified in Section 3.4.2.2 as: An x indicates that the lock modes for the two corresponding operations are either compatible or incompatible depending on whether the condition $x \notin R(S)$ and $x \notin R_I(S)$ (i.e. the nodes x are sieved out by the Predicate of S) holds for the location step.

The presented protocol XQUF-LP is a granular lock protocol. This protocol intoduces new lock modes SIL, SIF, SIT, SI, SA and SB. These modes are used to lock parent nodes of nodes inserted into the database. The inserted nodes themselves are locked in exclusive (X) lock mode. The X-lock mode is intended to distinguish between insert and delete operation. To prevent phantom problem the logical locks have to be introduced. XQUF-LP uses the same logical locks mechanism as Sedna Lock Protocol [48].

We employ the following lock modes in XQUF-LP:

- P lock mode (Pass-by mode). The P-lock is a shared lock designed for the Pass-by operation. In other words it is intended for mid-results of XPath location path. At the final location step of the path, P-locks on the destination nodes are eventually upgraded to S*, X or SR lock mode. P-locks are conditionally compatible with S*, X and XT lock modes.

- SI, SIL, SIF, SIT, SA and SB lock modes (Shared modes). These special shared locks are used by insert operations. These locks cover five insert functions: udp:insertInto (SI), upd:insertIntoAsLast (SIL), upd:insertIntoAsFirst (SIF), upd:insertAfter (SA), upd:insertAttributes (SIT) and upd:insertBefore (SB). For formal semantics of the functions see Subsection 4.6.4.

- X lock mode (Exclusive mode). The lock sets exclusive mode on a modified node. For instance, this lock is obtained for a newly created node.

- SR lock mode (Shared mode). The lock sets shared mode on a node's subtree. XPath queries require this kind of locks. Due to the semantics of XPath the results of the location path are the subtrees selected by the last location step. It implies the request of the SR (subtree read) lock for subtrees retrieved by location path.

- XT lock mode (Exclusive mode). The lock sets exclusive mode on a subtree. We use it for delete operations. The delete operation drops the subtree defined by location path.

- IS lock mode (Intention Shared mode). According to the granular locking protocol we have to obtain these locks on each ancestor of the node which is to be locked in a shared mode.

- IX lock mode (Intention Exclusive mode). According to the granular locking protocol we have to obtain these locks on each ancestor of the node which is to be locked in an exclusive mode.

Compatibility Matrix

The compatibility matrix is shown in Table 4.1. It contains twelve lock modes. The symbol + or - means that corresponding lock modes are compatible or incompatible. The symbol x means conditional compatibility of lock modes as described in previous paragraph.

Conversion Matrix

The conversion matrix of XQUF-LP is shown in Table 4.2. The rules contained in the matrix are applied if the same transactions holds a lock mode (granted) on the context

	granted											
requested	P	SI	SIL	SIF	SIT	SA	SB	X	SR	XT	IS	IX
P	+	+	+	+	+	+	+	x	+	x	+	+
SI	+	-	-	-	+	+	+	-	+	-	+	+
SIL	+	-	-	+	+	+	+	-	+	-	+	+
SIF	+	-	+	-	+	+	+	-	+	-	+	+
SIT	+	+	+	-	-	+	+	-	+	-	+	+
SA	+	+	+	+	+	-	+	-	+	-	+	+
SB	+	+	+	+	+	+	-	-	+	-	+	+
X	x	-	-	-	-	-	-	-	-	-	+	+
SR	+	+	+	+	+	+	+	-	+	-	+	-
XT	x	-	-	-	-	-	-	-	-	-	-	-
IS	+	+	+	+	+	+	+	+	+	-	+	+
IX	+	+	+	+	+	+	+	+	-	-	+	+

Table 4.1: XQUF-LP Compatibility Matrix

node and requests to lock this node by a new lock mode (requested). The resulting lock mode is the value of the corresponding cell. The special case are cells with a value $\bigcup$, which means that the both lock modes are acquired on the node.

	granted											
requested	P	SI	SIL	SIF	SIT	SA	SB	X	SR	XT	IS	IX
P	P	SI	SIL	SIF	SIT	SA	SB	X	SR	XT	IS	IX
SI	SI	SI	$\bigcup$	$\bigcup$	$\bigcup$	$\bigcup$	$\bigcup$	X	SI	XT	SI	IX
SIL	SIL	$\bigcup$	SIL	$\bigcup$	$\bigcup$	$\bigcup$	$\bigcup$	X	SIL	XT	SIL	IX
SIF	SIF	$\bigcup$	$\bigcup$	SIF	$\bigcup$	$\bigcup$	$\bigcup$	X	SIF	XT	SIF	IX
SIT	SIT	$\bigcup$	$\bigcup$	$\bigcup$	SIT	$\bigcup$	$\bigcup$	X	SIT	XT	SIT	IX
SA	SA	$\bigcup$	$\bigcup$	$\bigcup$	$\bigcup$	SA	$\bigcup$	X	SA	XT	SA	IX
SB	SB	$\bigcup$	$\bigcup$	$\bigcup$	$\bigcup$	$\bigcup$	SB	X	SB	XT	SB	IX
X	X	X	X	X	X	X	X	X	X	X	X	X
SR	SR	SI	SIL	SIF	SIT	SA	SB	X	SR	XT	SR	IX
XT	XT	XT	XT	XT	XT	XT	XT	XT	XT	XT	XT	XT
IS	IS	SI	SIL	SIF	SIT	SA	SB	X	SR	XT	IS	IX
IX	IX	IX	IX	IX	IX	IX	IX	X	IX	XT	IX	IX

Table 4.2: XQUF-LP Conversion Matrix

Granular Locking Protocol

The granular locking protocol defined in [23] has to generally fulfill these requirements:

1. Acquire locks from root to leaf.

2. Release locks from leaf to root.

3. To acquire an S mode or IS mode lock on a non-root node, one parent must be held in IS mode or higher.

4. To acquire an X, U, SIX, or IX mode lock on a non-root node, all parents must be held in IX mode or higher.

To achieve the highest throughput of running transactions we choose the lower bound [22] of accepted lock modes according the granular lock protocol requirements. To complete the formal semantics of XQuery Update Facility desribed in previous section we give the formal semantics of *Lock* function.

XQUF-LP protocol specific rules

- **Two-phase Locking Rule.** All lock modes, except P-locks [36], that are acquired or released must observe the two-phase locking protocol (2PL).

- **P-lock Rule.** Nodes in the location step are all locked by P-locks before performing the Node-Test and Predicate selection [36].

- **Upgrade Rules.**

  1. The P-locks on the nodes in the result set are upgraded to S*-locks before inserting nodes. The P-locks acquired on ancestors of nodes in the result set are converted to IS-locks. The X-lock is acquired on the insert node(s).

  2. The P-locks on the nodes in the result set are upgraded to SR- or X-locks before reading or writing. The P-locks acquired on ancestors of nodes in the result set are converted to IS-locks or IX-locks respectively.

  3. The P-locks on the nodes in the result set are upgraded to XT-locks before deleting them. The P-locks acquired on ancestors of nodes in the result set are converted to IX-locks.

- **Compatibility Rule.** A particular type of lock on a location step can be granted as long as the compatibility matrix is respected.

- **Release Rules**

1. S*-, X-, SR-, XT-, IS- or IX-locks can only be released in the shrinking phase of a transaction. That is, releasing them must observe the two-phase locking rule.
2. P-locks on the nodes which are sieved out by the predicate or node-test operation are released after location step finishes.

Lock Function Semantics

NTL (Nodes to Lock) is a stack.

The Semantics of A $\hookleftarrow$ B is push B on top of stack A.

$$\frac{el == \epsilon}{Lock(el, lock, g, l) = LockReq(g, l)}$$

$$\frac{el \neq \epsilon}{Lock(el, lock, g, l) = Lock(el.par(), lock.par(), g, l[NTL \hookleftarrow < el, lock >])}$$

$$\frac{isComp(< el, lock >, g, l[NTL == << el, lock >: rest >)) == (true, \epsilon)}{\begin{array}{l} LockReq(g, l[NTL == << el, lock >: rest >]) = \\ \quad = LockReq(addToLocks(g, < el, lock, l[TRANS] >), l[NTL := < rest >]) \end{array}}$$

$$addToWFG : \mathcal{G}_{cont} \times Transaction \times Transaction \longrightarrow \mathcal{G}_{cont}$$

$$addToWFG(g, t_1, t_2) = g[WFG := addEdge(g[WFG], t_1, t_2)]$$

$$\frac{isComp(< el, lock >, g, l[NTL == << el, lock >: rest >)) == (false, wtrans)}{\begin{array}{l} LockReq(g, l[NTL == << el, lock >: rest >) = \\ \quad = Wait(addToWFG(g[WAIT \leftarrow < el, lock, l[TRANS] >], l[TRANS], wtrans), \\ \hfill l[NTL == < rest >]) \end{array}}$$

$$isWaiting : WFG \times Transaction \longrightarrow Bool$$

$$isWaiting(w, t) = \begin{cases} true, & \text{if } \exists t_x : Transaction | (t, t_x) \in w \\ false, & otherwise \end{cases}$$

$$CheckDeadlock : \mathcal{G}_{cont} \times \mathcal{L}_{cont} \longrightarrow \mathcal{G}_{cont} \times \mathcal{L}_{cont}$$

$$CheckDeadlock(g,l) = \begin{cases} Wait(g,l), & \text{if } deadlock(g[WFG]) == false \\ abortTransaction(g,l), & otherwise \end{cases}$$

$$\frac{isWaiting(g[WFG], l[TRANS]) == true}{Wait(g,l) = CheckDeadlock(g,l)}$$

$$\frac{isWaiting(g[WFG], l[TRANS]) == false}{\begin{array}{r} Wait(g[WAIT \leftarrow < el, lock, l[TRANS] >], l[NTL == < rest >]) = \\ = LockReq(g, l[NTL := << el, lock >: rest >]) \end{array}}$$

$$LockReq(g, l[NTL == < \epsilon >]) = (g,l)$$

Theorem 4.7.1. *Transaction histories generated by the XQUF-LP protocol are serializable.*

Proof. To prove this theorem we assume READ and UPDATE operations defined in definition 3.3.4. By definition 2.1.6 a serializable history over a set S of committed transactions is a history whose effect on any consistent database instance is guaranteed to be equivalent to some serial history over S.

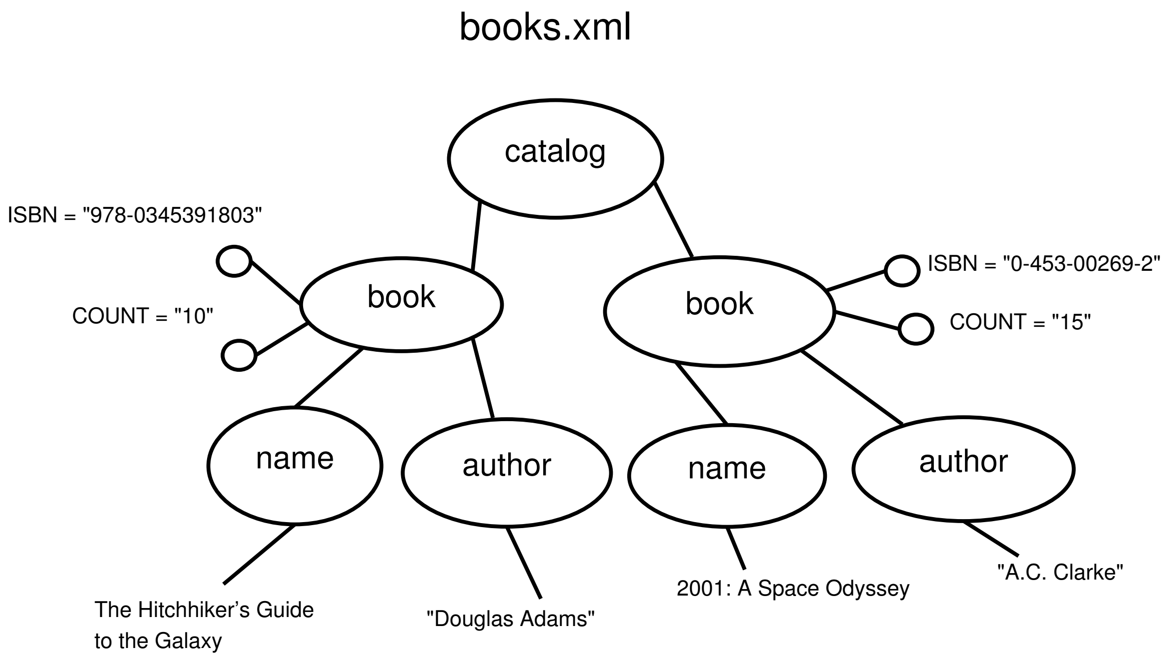


Figure 4.4: XML document

1. **One transaction.** If we have only one transaction T running in the system then it forms a serializable history. It is equivalent to a serial history containing only T.

2. **Two transactions.** In case there are two transactions we have to distinguish among six cases (type of conflicts):

   - R-W conflict on the same node. $<< T1, R, (o, 1) >, < T2, U, (o, 2) >, < T1, R, (o, 2) >>$. Transaction *T1* reads object o, then transaction *T2* writes into object o value 2, followed by the read of object o by transaction *T1*. For example the object o is a node "author" with value "Douglas Adams" in Figure 4.4. Then the history generated by the XQUF-LP is:

$<< T1, BEGIN >$,
$< T2, BEGIN >$,
$< T1, LOCK(ancestors(o), IS) >$,
$< T1, LOCK(o, SR) >$,
$< T1, READ(o) >$,
$< T2, LOCK(ancestors(o), IX) >$,
$< T2, LOCK(o, X) >$, at this point a transaction *T2* is postponed, because *X-lock* is not compatible with *SR-lock*, until the transaction *T1* ABORTs or COMMITs. Thus, the R-W conflict is solved by the protocol. The generated history by the protocol is:
$<< T1, BEGIN >$,
$< T2, BEGIN >$,
$< T1, LOCK(ancestors(o), IS) >$,
$< T1, LOCK(o, SR) >$,
$< T1, READ(o) >$,
$< T2, LOCK(ancestors(o), IX) >$,
$< T1, LOCK(ancestors(o), IS) >$,
$< T1, LOCK(o, SR) >$,
$< T1, COMMIT >$,
$< T2, LOCK(o, X) >$,
$< T2, UPDATE(o, 2) >$,
$< T2, COMMIT >>$.
This history is in accordance with a serial history $< T1, T2 >$.

- W-R conflict on the same node. $<< T2, U, (o, 2) >, < T1, R, (o, 2) >, < T2, U, (o, 3) >>$. Transaction *T2* writes object *o*, then transaction *T1* reads object *o*, followed by the write of a new value to the object *o* by transaction *T2*. For example the object *o* is a node "author" with value "Douglas Adams" in Figure 4.4. Then the history generated by the XQUF-LP is:
$<< T1, BEGIN >$,
$< T2, BEGIN >$,
$< T2, LOCK(ancestors(o), IX) >$,

$< T2, LOCK(o, X) >$,

$< T2, UPDATE(o, 2) >$,

$< T1, LOCK(ancestors(o), IS) >$,

$< T1, LOCK(o, SR) >$, at this point a transaction $T1$ is post-poned, because *X-lock* is not compatible with *SR-lock*, until the transaction $T2$ ABORTs or COMMITs. Thus, the W-R conflict is solved by the protocol. The generated history by the protocol is:

$<< T1, BEGIN >$,

$< T2, BEGIN >$,

$< T2, LOCK(ancestors(o), IX) >$,

$< T2, LOCK(o, X) >$,

$< T2, UPDATE(o, 2) >$,

$< T1, LOCK(ancestors(o), IS) >$,

$< T2, LOCK(o, SR) >$,

$< T2, LOCK(ancestors(o), IX) >$,

$< T2, LOCK(o, X) >$,

$< T2, UPDATE(o, 3) >$,

$< T2, COMMIT >$,

$< T1, LOCK(o, SR) >$,

$< T1, COMMIT >>$.

This history is in accordance with a serial history $< T2, T1 >$.

- W-W conflict on the same node. $<< T2, R, (o, 1) >, < T1, U, (o, 2) >, < T2, U, (o, 3) >>$. Transaction *T2* reads object o, then transaction *T1* writes object o, followed by the write of a new value to the object o by transaction *T2*. For example the object o is a node "author" with value "Douglas Adams" in Figure 4.4. Then the history generated by the XQUF-LP is:

  $<< T1, BEGIN >$,

  $< T2, BEGIN >$,

  $< T2, LOCK(ancestors(o), IS) >$,

  $< T2, LOCK(o, SR) >$,

  $< T2, READ(o) >$,

$< T1, LOCK(ancestors(o), IX)$,

$< T1, LOCK(o, X) >$, at this point a transaction $T1$ is post-poned, because *X-lock* is not compatible with *SR-lock*, until the transaction $T2$ ABORTs or COMMITs. Thus, the W-R conflict is solved by the protocol. The generated history by the protocol is:

$<< T1, BEGIN >$,

$< T2, BEGIN >$,

$< T2, LOCK(ancestors(o), IS) >$,

$< T2, LOCK(o, SR) >$,

$< T2, READ(o) >$,

$< T1, LOCK(ancestors(o), IX) >$,

$< T2, LOCK(ancestors(o), IX) >$,

$< T2, LOCK(o, X) >$,

$< T2, UPDATE(o, 3) >$,

$< T2, COMMIT >$,

$< T1, LOCK(o, X) >$

$< T1, UPDATE(o, 2) >$

$< T1, COMMIT >>$.

This history is in accordance with a serial history $< T2, T1 >$.

- R-W conflict in the subtree. $o1 \in ancestors(o2) << T2, R, (o1) >, < T1, U, (o2, 1) >$ $, < T2, R, (o1) >>$. Transaction *T2* reads object *o1*, then transaction *T1* writes value 1 into object *o2*, followed by the read of object *o1* by transaction *T2*. For example the object *o1* is node "book" with attribute "count=10" and the object *o2* is node "author" with value "Douglas Adams" in Figure 4.4. Then the history generated by the XQUF-LP is:

  $<< T1, BEGIN >$,

  $< T2, BEGIN >$,

  $< T2, LOCK(ancestors(o1), IS) >$,

  $< T2, LOCK(o1, SR) >$,

  $< T2, READ(o1) >$,

  $< T1, LOCK(ancestors(o2), IX) >$,, at this point transaction *T1* is post-poned, because *IX-lock* is not compatible with *SR-lock*, until the transaction

T2 ABORTs or COMMITs. Thus, the R-W conflict is solved by the protocol. The generated history by the protocol is:
$<< T1, BEGIN >$,
$< T2, BEGIN >$,
$< T2, LOCK(ancestors(o1), IS) >$,
$< T2, LOCK(o1, SR) >$,
$< T2, READ(o1) >$,
$< T2, LOCK(ancestors(o1), IS) >$,
$< T2, LOCK(o1, SR) >$,
$< T2, READ(o1) >$,
$< T2, COMMIT >$,
$< T1, LOCK(ancestors(o2), IX) >$,
$< T1, LOCK(o2, X) >$
$< T1, UPDATE(o2, 1) >$
$< T1, COMMIT >>$
This history is in accordance with a serial history $< T2, T1 >$.

- W-R conflict in the subtree. $o1 \in ancestors(o2) << T2, U, (o2, 2) >, < T1, R, (o1) >, < T2, U, (o2, 1) >>$. Transaction *T2* reads object *o2*, then transaction *T1* deletes object *o2*, followed by the read of object *o2* by transaction *T2*. For example the object *o1* is node "book" with attribute "count=10" and the object *o2* is node "author" with value "Douglas Adams" in Figure 4.4. Then the history generated by the XQUF-LP is:
$<< T1, BEGIN >$,
$< T2, BEGIN >$,
$< T2, LOCK(ancestors(o2), IX) >$,
$< T2, LOCK(o2, X) >$,
$< T2, UPDATE(o2, 2) >$,
$< T1, LOCK(ancestors(o1), IS) >$,
$< T1, LOCK(o1, SR) >$,
$< T1, READ(o1) >$,
$< T2, LOCK(ancestors(o2), IX) >$, at this point transaction *T2* is postponed, because *IX-lock* is not compatible with *SR-lock*, until the transaction

T1 ABORTs or COMMITs. Thus, the W-R in the subtree conflict is solved by the protocol. The generated history by the protocol is:
$<< T1, BEGIN >$,
$< T2, BEGIN >$,
$< T2, LOCK(ancestors(o2), IX) >$,
$< T2, LOCK(o2, X) >$,
$< T2, UPDATE(o2, 2) >$,
$< T1, LOCK(ancestors(o1), IS) >$,
$< T1, LOCK(o1, SR) >$,
$< T1, READ(o1) >$,
$< T1, COMMIT >$,
$< T2, LOCK(ancestors(o2), IX) >$,
$< T2, LOCK(o2, X) >$,
$< T2, UPDATE(o2, 1) >$,
$< T2, COMMIT >>$.
This history is in accordance with a serial history $< T1, T2 >$.

- W-W conflict in the subtree. $o1 \in ancestors(o2)$ $<< T2, R, (o2) >, < T1, U, (o1, 1) >, < T2, U, (o2, 2) >>$. Transaction *T2* reads object *o2*, then transaction *T1* writes object *o1*, followed by the write of object *o2* by transaction *T2*. For example the object *o1* is node "book" with attribute "count=10" and the object *o2* is node "author" with value "Douglas Adams" in Figure 4.4. Then the history generated by the XQUF-LP is:
  $<< T1, BEGIN >$,
  $< T2, BEGIN >$,
  $< T2, LOCK(ancestors(o2), IS) >$,
  $< T2, LOCK(o2, SR) >$,
  $< T2, READ(o2) >$,
  $< T1, LOCK(ancestors(o1), IX) >$, at this point transaction *T2* is postponed, because *IX-lock* is not compatible with *SR-lock*, until the transaction *T1* ABORTs or COMMITs. Thus, the D-W in the subtree conflict is solved by the protocol. The generated history by the protocol is:
  $<< T1, BEGIN >$,

$< T2, BEGIN >$,
$< T2, LOCK(ancestors(o2), IS) >$,
$< T2, LOCK(o2, SR) >$,
$< T2, READ(o2) >$,
$< T2, LOCK(ancestors(o2), IX) >$
$< T2, LOCK(o2, X) >$,
$< T2, UPDATE(o2, 2) >$,
$< T2, COMMIT >$,
$< T1, LOCK(ancestors(o1), IX) >$,
$< T1, LOCK(o1, X) >$,
$< T1, UPDATE(o1, 1) >$,
$< T1, COMMIT >$.
This history is in accordance with a serial history $< T2, T1 >$.

3. **More than two transactions.** Every dependency in the history of more than two transactions can be reduced on R-W, W-W or W-R conflict between two transactions according to proof by Gray [23].

□

The algorithm describing this protocol is shown in Figure 4.5.

Phantom Prevention

In Section 2.1.5.1 we introduced phantom read in databases. As we mentioned in paragraph about XDGL and logical locks 3.4.2.1 to achieve high concurrency we also have to introduce some kind of predicate or logical locks into our locking protocol. Our protocol uses predicate locks to prevent phantom reads. For example when the node is inserted then the predicate lock of the form *lock(IN, node-name, transaction)* is acquired on the parent node of the inserted node. When reading nodes then the predicate lock of the form *lock(L, node-name, transaction)* is acquired on the parent node of the node which is being read. These locks are compatible if and only if *node-name*s differs.

XQUF-LP and Degrees of Isolation

In Section 2.1.5 we defined isolation levels according to lock protocol features. In previous sections we assumed that the protocols conforms to ACID properties (Degree 3 without

```
input:  CN - context node
        LM - lock mode instance of LockMode class
        t - transaction

lockRequest(CN, LM, t) {                    // request a lock mode
  if(isCompatible(LM, CN.getLock()) {       // if LM is compatible
    lock(CN, LM);                           // assign it
  } else {
    suspend(t);                             // suspend transaction
    exit();                                 // do not continue
  }
}

getParents(CN, LM) {
  parents:=new Stack();
  while(CN.getParent()!=null){             // while exists parent
    parents.add(<CN.getParent(), LM.getParentLockType()>);
    CN = CN.getParent();
    LM = LM.getParentLockType();
  }
  return parents;
}

parents:=getParents(CN, LM);
while(!parents.empty()) {
  parent_lock:=parents.pop()
  lockRequest(parent_lock.first(),
              parent_lock.second());        // request lock modes
}
```

Figure 4.5: Granular Locking Protocol Algorithm

phantoms). However in many applications this strict degree of isolation is not needed for their correct functioning. We also considered these needs the XQUF-LP protocol design. As the result the protocol XQUF-LP can be very easily tailored to conform each of the mentioned isolation levels. The XQUF-LP can work in certain degree of isolation if fulfills the requirements given in Section 2.1.5.

4.7.4 Semantics Evaluation Example

In this section we show the semantics execution on the following XQUF query:

```
BEGIN, replace node fn:doc("bib.xml")/books/book[1]/publisher
        with fn:doc("bib.xml")/books/book[2]/publisher,
COMMIT
```

First, we rewrite the previous query into this semantics expression:

$$[\![COMMIT]\!]([\![XQUF_{query}]\!]([\![BEGIN]\!](g, l := \epsilon)))$$

We assume that this is the first transaction in the system with the beginning state:

$$g = gcont(document(\text{"}bib.xml\text{"}, element(\text{"}books\text{"}, ...)), (), \emptyset, (), \emptyset, emptyWFG)$$

We rewrite the $[\![BEGIN]\!]$ expression on:

$$\begin{aligned}
&[\![COMMIT]\!]([\![XQUF_{query}]\!]([\![BEGIN]\!](g, l == \epsilon))) = \\
&= [\![COMMIT]\!]([\![XQUF_{query}]\!](beginTransaction(g, l, createTrans(beginTrans())))) = \\
&= [\![COMMIT]\!]([\![XQUF_{query}]\!](g[TRANS \leftarrow trans(1)], l[TRANS := trans(1)]))
\end{aligned}$$

The transaction $trans(1)$ is ready and stored in the global state g and the local state l. The next step is the evaluation of the replace expression $[\![XQUF_{query}]\!]$. In next equations we omit $[\![COMMIT]\!]$ expression and internal variables of g and l states for better readability.

TE = fn:doc("bib.xml")/books/book[1]/publisher

ES = fn:doc("bib.xml")/books/book[2]/publisher

$$\begin{aligned}
&[\![replace\, node\, TE\, with\, ES]\!]_{xquf}(g, l) = \\
&\quad = CC(g[PUL \leftarrow \{u : rN([\![TE]\!]_{xque}(g, l), [\![ES]\!]_{xque}(g, l))\}, \\
&\qquad CCL \leftarrow \{< replace, [\![TE]\!]_{xque}(g, l), [\![ES]\!]_{xque}(g, l) >\}], l)
\end{aligned}$$

First we show the evaluation of the first expression $[\![TE]\!]$.

$$[\![TE]\!]_{xque}(g,l) = [\![\text{fn:doc("bib.xml")/books/book[1]/publisher}]\!]_{xque}(g,l)$$

$$\begin{aligned}
&[\![\text{fn:doc("bib.xml")/books/book[1]/publisher}]\!]_{xque}(g,l) = \\
&= [\![\text{child::publisher}]\!]([\![RelPath]\!](g,l)) = \\
&= [\![\text{child::publisher}]\!](([\![\text{child::book[1]}]\!])[\![RelPath]\!](g,l)) = \\
&= [\![\text{child::publisher}]\!](([\![\text{child::book[1]}]\!])[\![child :: books]\!][\![fn : doc(\text{"}bib.xml\text{"})]\!](g,l)) = \\
&= [\![\text{child::publisher}]\!](([\![\text{child::book[1]}]\!])[\![child :: books]\!](g, l[RES := doc(\text{"}bib.xml\text{"})]) = \\
&= ...(([\![\text{child::book[1]}]\!])(\text{node-test}(\lambda\text{"}books\text{"}, axis(\text{"}child\text{"}, doc(\text{"}bib.xml\text{"}), g, l) = \\
&= ...(\text{node-test}(\lambda\text{"}books\text{"}, Lock(el(\text{"}books\text{"}, ...), P, g, l[RES \leftarrow el(\text{"}books\text{"}, ...)]) = \\
&= ...(\text{node-test}(\lambda\text{"}books\text{"}, el(\text{"}books\text{"}, ...), g[LOCKS \leftarrow < el(\text{"}books\text{"}), P, trans(1) >], l) = \\
&= ...(([\![\text{child::book[1]}]\!])(g, l[RES := (el(\text{"}books\text{"}))]) = \\
&= ...(\text{fs:item-at}([\![\text{child::book}]\!](g, l[RES := (el(\text{"}books\text{"}))], 1)) = \\
&= ...(\text{fs:item-at}(g[LOCKS \leftarrow (< el(\text{"}book\text{"}_1), P, trans(1) >, \\
&\qquad\qquad\qquad\qquad\qquad < el(\text{"}book\text{"}_2), P, trans(1) >)], \\
&\qquad\qquad l[RES := (el(\text{"}book\text{"}_1), el(\text{"}book\text{"}_2))], 1)) = \\
&= [\![\text{child::publisher}]\!](g, l[RES := (el(\text{"}book\text{"}_1)]) = \\
&... \\
&= (g[LOCKS := \{< el(\text{"}book\text{"}_1), P, trans(1) >, < el(\text{"}book\text{"}_2), P, trans(1) >, \\
&\qquad\qquad < el(\text{"}books\text{"}), P, trans(1) >, < el(\text{"}publisher\text{"}_1), P, trans(1) >, \\
&\qquad\qquad < doc(\text{"}bib.xml\text{"}), P, trans(1) >\}], \\
&\quad l[RES := (el(\text{"}publisher\text{"}_1))])
\end{aligned}$$

Finally the *LockRead* function is applied as XQuery Semantics specification states. It implies that the final state is:

$$\begin{aligned}(g[LOCKS == \{&< el(\text{"}book\text{"}_1), IS, trans(1) >, < el(\text{"}book\text{"}_2), P, trans(1) >,\\ &< el(\text{"}books\text{"}), IS, trans(1) >, < el(\text{"}publisher\text{"}_1), SR, trans(1) >,\\ &< doc(\text{"}bib.xml\text{"}), IS, trans(1) >\}],\\ l[RES == (el(\text{"}publisher\text{"}_1))])&\end{aligned}$$

The evaluation of $[\![ES]\!]$ is obviously similar. We show only the result here.

$$[\![ES]\!]_{xque}(g,l) = [\![\text{fn:doc("bib.xml")/books/book[2]/publisher}]\!]_{xque}(g,l)$$

$$\begin{aligned}&[\![ES]\!]_{xque}(g,l) =\\ &= (g[LOCKS := \{< el(\text{"}book\text{"}_1), IS, trans(1) >, < el(\text{"}book\text{"}_2), IS, trans(1) >,\\ &\qquad < el(\text{"}books\text{"}), IS, trans(1) >, < el(\text{"}publisher\text{"}_1), SR, trans(1) >,\\ &\qquad < doc(\text{"}bib.xml\text{"}), IS, trans(1) >, < el(\text{"}publisher\text{"}_2), SR, trans(1) >\}],\\ &\quad l[RES := (el(\text{"}publisher\text{"}_2))])\end{aligned}$$

So, the evaluation of the original expression after the evaluations shown above is:

$$\begin{aligned}&[\![replace\ node\ TE\ with\ ES]\!]_{xquf}(g,l) =\\ &\quad = CC(g[PUL \leftarrow \{\text{u:rN}([\![TE]\!]_{xque}(g,l), [\![ES]\!]_{xque}(g,l))\},\\ &\qquad CCL \leftarrow \{< replace, [\![TE]\!]_{xque}(g,l), [\![ES]\!]_{xque}(g,l) >\}], l) =\\ &\quad = CC(g[PUL \leftarrow \{\text{u:rN}(el(\text{"}publisher\text{"}_1), el(\text{"}publisher\text{"}_2))\},\\ &\qquad CCL \leftarrow \{< replace, el(\text{"}publisher\text{"}_1), el(\text{"}publisher\text{"}_2))\}\}], l)\end{aligned}$$

In the next step the Constraints Checker checks the results of expressions for errors according to semantics described in Section 4.6.3. In this case no error is found, hence the evaluation can continue:

$$
\begin{aligned}
&CC(g[PUL \leftarrow \{\text{u:rN}(el(\text{"}publisher\text{"}_1), el(\text{"}publisher\text{"}_2))\},\\
&\quad CCL \leftarrow \{< replace, el(\text{"}publisher\text{"}_1), el(\text{"}publisher\text{"}_2))\}\}], l) =\\
&= \text{upd:applyUpdates}(g[PUL], \text{"}strict\text{"}, false, g, l) =\\
&= \text{u:rN}(el(\text{"}publisher\text{"}_1), el(\text{"}publisher\text{"}_2), g, l) =\\
&= Lock(el(\text{"}publisher\text{"}_1), XT, g, l)
\end{aligned}
$$

The *Lock* function with XT lock mode is applied to the element *publisher* which is being replaced. Because the element *publisher* is already locked, its lock must be upgraded according to conversion matrix rules. After that step the final state is:

$$
\begin{aligned}
&(g[LOCKS == \{< el(\text{"}book\text{"}_1), IX, trans(1) >, < el(\text{"}book\text{"}_2), IS, trans(1) >,\\
&\qquad < el(\text{"}books\text{"}), IX, trans(1) >, < el(\text{"}publisher\text{"}_1), XT, trans(1) >,\\
&\qquad < doc(\text{"}bib.xml\text{"}), IX, trans(1) >, < el(\text{"}publisher\text{"}_2), SR, trans(1) >\},\\
&\quad TRANS = \{trans(1)\}],\\
&\ l[TRANS == trans(1)])
\end{aligned}
$$

The last evaluation step is COMMIT of the transaction. In this step all locks held by the transaction are released:

$$
\begin{aligned}
&[\![COMMIT]\!](g, l[TRANS == t]) = commitTransaction(g, l) =\\
&= (g[TRANS := TRANS \setminus \{t\}, LOCKS := LOCKS \setminus \{< node, lock, t >\}],\\
&\quad l[TRANS := \epsilon])
\end{aligned}
$$

4.8 Semantics Verification

In the previous section we introduced the formal specification of the XQuery Update Facility language semantics. In this section we verify this specification by its implementation in Maude language [15].

Maude is a language supporting executable specification and declarative programming in rewriting logic. Rewriting logic is very well suited for our purposes. It is a logical framework that allows concurrency [42] and can be used to implement executable specification (semantics). Verdejo and Marti-Oliet in [65] successfuly implemented CCS[8] operational semantics in Maude. Maude's rewriting logic is expressive enough[9] to allow implementing XQUF semantics.

As a result of our experiments in semantics verification we developed two different implementations. The first implementation uses functional modules and equational modules. This implementation has the important disadvantage. It does not allow to pass global state to more than one subexpression. For better explanation consider Figure 4.6. In this Figure we are passing the same global state to $[\![TE]\!]_{xque}(g,l)$, $[\![ES]\!]_{xque}(g,l)$ and CC. All of these functions modifies the global state g. If we evaluate them concurrently we get a new global state g' from each evaluation. That is wrong. We need to refer to the same global state g. To solve this problem we used Full Maude library and its object modules to implement the semantics. Full Maude is built on Maude's powerful reflection and metaprogramming capabilities. We model the semantics as a concurrent object system. We had to solve many difficulties during the implementation of the semantics into Maude. The first obvious thing is that the concurrent object system can be viewed as a configuration which consists of objects and messages at the beginning. These message are applied to corresponding objects "concurrently". But we need to preserve the order of operations inside the transaction. We had to develop a technique to preserve it. This technique is based on the stack structure which is unique for each transaction. Using it we are able to preserve the order of operations inside the transaction.

$$
\begin{aligned}
&[\![replace\, node\, TE\, with\, ES]\!]_{xquf}(g,l) = \\
&\qquad = CC(g[PUL \leftarrow \{\text{u:rN}([\![TE]\!]_{xque}(g,l),[\![ES]\!]_{xque}(g,l))\}, \\
&\qquad\qquad CCL \leftarrow \{< replace, [\![TE]\!]_{xque}(g,l),[\![ES]\!]_{xque}(g,l) >\}],l)
\end{aligned}
$$

Figure 4.6: Replace Expression Semantics

[8]CCS is Milner's Calculus of Communicating Systems, which models asynchronous communication of processes.

[9]It allows concurrency.

4.8.1 XQUF-LP Framework

As a result of our experiments the XQUF-LP framework was built as a tool for proving the correctness of locking protocols. XQUF-LP framework can be easily tailored for needs of the verificated locking protocol. The framework has a modular architecture shown in Figure 4.8.1[10]. It consists of five modules: database, xpath, xdm, transactions and xquf. Each modules is seated in its own diretory.

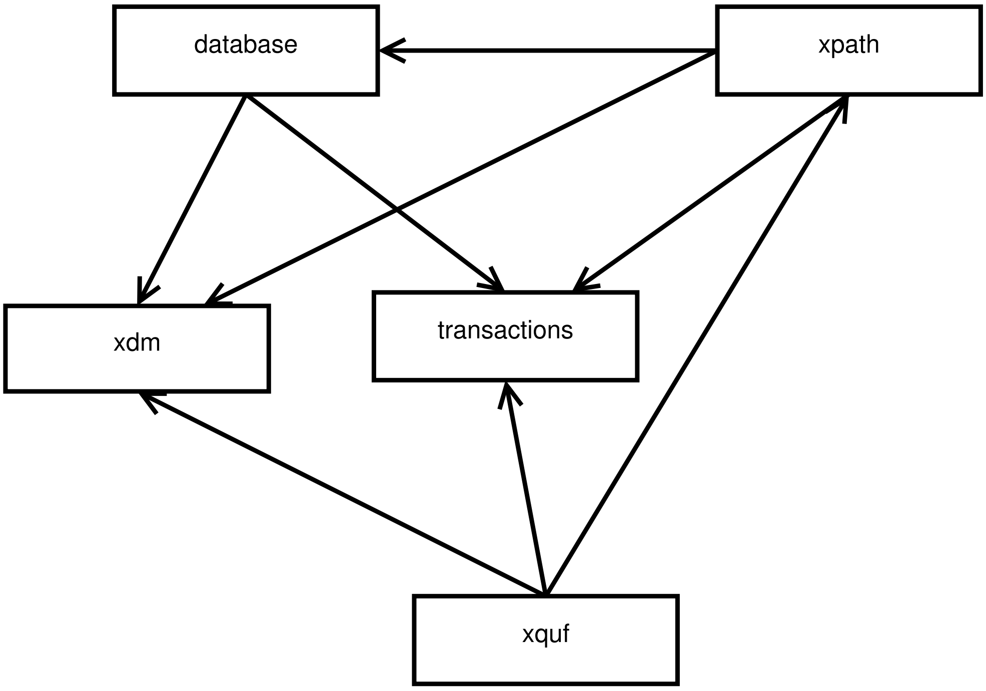


Figure 4.7: Module dependency in XQUF-LP framework

The XDM module implements the lite version of XPath and XQuery data model as presented in Appendix B. The module functionality is wrapped in XDM-ALL functional module. A small example of the specification written in Maude is shown in Figure 4.8.1. The figure lists the code of the XPath-Base Module. The framework is not fully implemented but we experimentally verified that the idea is correct. The future research will be focused on the completion of the framework to fully support proposed semantics. It will be the subject of the bachelor's and master's thesis at our department. The source code of the framework can be found in [57].

[10]The arrow from module A to module B means that A depends on B.

```
(fmod XPATH-AXES-CONST is
  sort AxisType .
  op self : -> AxisType [ctor] .
  op child : -> AxisType [ctor] .
  op attribute : -> AxisType [ctor] .
  op parent : -> AxisType [ctor] .
endfm)

(omod XPATH-BASE is
  pr XDM-ALL .
  pr DATABASE .
  pr XPATH-AXES-CONST .
  pr LOCK-TABLE-XQUF .
  pr CONVERSION .

  sorts Expr XPathExpr StepExpr ForwardStep ReverseStep .
  subsort XPathExpr < Expr .
  subsort Node < XPathExpr .
  subsort StepExpr < XPathExpr .
  subsort ForwardStep ReverseStep < XPathExpr .
  sort Axis .

  op doc : Uri -> StepExpr [ctor] .
  op axis : AxisType Qid  -> StepExpr [ctor] .

  op _/_ : XPathExpr StepExpr -> XPathExpr [ctor] .
  op _/_ : StepExpr StepExpr -> XPathExpr [ctor ditto] .

  msg beginTrans : Oid Expr -> Msg .
  msg `(_`,_`)`[`[_`]`] : Oid Nat Expr -> Msg .

  msg mnode : Oid Node Nat -> Msg .
  msg maxis : Oid AxisType Node Qid Nat -> Msg .
  msg mnodetest : Oid Qid Node Nat -> Msg .
  msg mdoc : Oid Uri Nat -> Msg .
  msg lock : Oid LockMode Node Nat -> Msg .
endom)
```

Figure 4.8: XPath-Base Module

5 Benchmarking

In this chapter we first give a brief introduction to the benchmarking of XML databases. In section 5.2 we present our benchmark proposal to measure transaction manager's performance. The results of this chapter were published in [61].

Many of native XML database engines do not either support transactional processing on the user level (eXist, Xindice) or they support it only partially (Berkeley DB, Sedna). So, there are only a few native XML engines that have ambitions to fully implement a node-level locking mechanism.

On the other hand, there are complex, application-based benchmarks that care about transactions (see mainly TPoX bellow). But the main aim of this chapter is to present the transaction manager benchmark which is simple enough to implement and use.

5.1 XML Application Benchmarks Overview

In the following paragraphs, we provide a very brief description of several XML benchmarks. More details about their data models, queries, etc. can be found in [13].

5.1.1 X007 Benchmark

This benchmark was developed upon the 007 benchmark – it is an XML version of 007, only enriched by new elements and queries for specific XML related testing.

Similarly to the 007 benchmark, it provides 3 different data sets: small, intermediate and large. The majority of 007 queries is focused on document oriented processing in object oriented DBs. The X007 testing set [8] is divided into three groups:

- traditional database queries,
- data navigation queries, and
- document oriented queries.

Data manipulation and transactional processing are not considered in X007. A good example of the application of this benchmark can be found in [9].

5.1.2 XMark Benchmark

XMark benchmark [53] simulates an internet auction and consists of 20 queries. The main entities are an item, a person, an opened and finished auction, and a category. Items represent either an object that has already been sold or an offered object. Persons have subelements such as a name, e-mail, telephone number, etc. Category, finally, includes a name and a description. The data included in the benchmark is a collection of 17,000 most frequently used words in Shakespeare's plays. The standard size of the document is 100MB. This size, then, is taken as 1.0 on a scale. A user can change the size of the data up to ten times from the default.

5.1.3 XMach-1

XMach-1 benchmark [6] is based on a web application and considers a different sets of XML data with a simple structure and a relatively small size. XMach-1 supports data with or without defined structure. A basic measure unit is XQps – a number of XML queries per second.

The benchmark architecture consists of four parts: the XML database, an application server, clients for data loading and clients for querying. The database has a folder based structure and XML documents are designed to be loaded (by a load client) from various data sources located in the internet. Each document has a unique URL maintained (together with metadata) in a folder based structure. Furthermore, an application server keeps a web server and other middleware components for XML documents processing and for an interaction with a backend database.

Each XML file represents an article with elements such as a name, a chapter, a paragraph, etc. Text data are taken from a natural language. A user can change the XML file size by changing the quantity of the article elements. By changing the quantity of XML files the size of the data file is controlled. XMach-1 assumes that the size of data files is small (1-14kB).

XMach-1 evaluates both standard and non-standard language features, such as insert, delete, URL query and aggregation functions. The benchmark consists of 8 queries and 2 update operations. The queries are divided into 4 groups according the the common characteristics they portray:

- group 1: simple selection and projection with a comparison of elements or attributes
- group 2: it requires the use of element order
- group 3: tests aggregation capabilities and it uses metadata information
- group 4: tests operation updates

5.1.4 TPoX

Transaction Processing over XML [44] is an application benchmark that simulates financial applications. It is used to evaluate the efficiency of XML database systems with a special attention paid to XQuery, SQL/XML, XML storage, XML indexing, XML scheme support, XML update, logging and other database aspects. It appears to be the most complex one and it also is the best contemporary benchmark.

The benchmark simulates on-line trading and uses FIXML to model a certain part of the data. FIXML is an XML version of FIX (Financial Information eXchange): a protocol used by the majority of leading financial companies in the world. FIXML consists of 41 schemes which, in turn, contain more than 1300 type definitions and more than 3600 elements and attributes.

TPoX has 3 different types of XML documents: Order, Security, and CustAcc which includes a customer with all her accounts. The information about holdings is included in the account data. Order documents follow the standard FIXML schema. Typical document sizes are following: 3 - 10 KB for Security, 1 - 2 KB for Order, and 4 - 20 KB for combined Customer/Account documents.

To capture the diversity/irregularity often seen in real-world XML data, there are hundreds of optional attributes and elements with typically only a small subset present in any given document instance (such as in FIXML) [43].

5.1.5 Framework TaMix for XML Benchmarks

TaMix [25] is a framework that provides an automated runtime environment for benchmarks on XML documents. It is mainly developed at the University of Kaiserslautern. Those benchmarks consist of a specified amount of update operations per transaction. It is a simulation of a bank application tailored to update operations. Unfortunately, the framework's more detailed specification is not publicly available. Therefore we were not able to implement it in our environment. Instead we used the idea of this framework for our benchmark's implementation.

5.1.6 XML Application Benchmarks – Summary

The benchmarks XMark and X007 can be viewed as combined or composite: their data and queries are in fact fictious application scenarios, but, at the same time, they try to test essential components of the languages – XQuery and XPath. On the other hand they ignore update operations.

XMach-1 and TPoX benchmarks consider both queries as well as updates. Hence, these benchmarks seems much more relevant to our needs. Unfortunately, both benchmarks use very complicated data models and their implementation takes a lot of time. TaMix framework seems suitable for our implementation but there is no detailed description available.

5.2 Performance Benchmarking

In the beginning of our research we asked the question "How to measure a transaction manager's performance?". The main motivation was to measure the performance of the Transaction Manager module implemented in the CellStore.

We found that there are two possibilities for measuring Transaction Manager's performance. The first possibility is to measure the performance of the whole database system twice. A first measurement is performed with a transaction manager involved and a second measurement without a transaction manager. The advantage of this possibility is an easier realisation of measurement but it does not provide optimal results because it is influenced by the rest of the database system.

The second possibility is based on separating the transaction manager from the database system. The important advantage of this possibility lies in providing more relevant results. Disadvantage of this approach is that the designer of the database system has to think about the modularity at a design time.

Our approach for measuring the performance can be applied in both cases. Finally, we decided to design a simple benchmark to get a general overview of the Transaction Manager's performance.

File Name	**G's Factor**	**Size**
db001.xml	0.01	1 154 kB
db005.xml	0.05	5 735 kB
db01.xml	0.1	11 596 kB
db02.xml	0.2	23 364 kB

Table 5.1: Database sizes depend on Generator's Factor

5.3 Benchmark specification

Our benchmark specification generally consists of

- the XML Schema of a test database
- sizes of database instances
- benchmarked operations (queries and updates)
- output consists of a duration of benchmarked operations in milliseconds.

We chose XMark's database model schema [54] as the schema for our test database. This schema covers our requirement of a real world application schema for online transaction processing. It is based on the model of internet auctions. XMark also includes a generator for database instances. Then it can be easily adjusted to another testing environment. Our benchmark uses 4 different sizes of a test database. In Table 5.1 are described database instances that the benchmark uses. The generator's factor is a *scaling factor* f for the XMark generator.

The benchmark's tests are described in Table 5.2. Tests 1 and 2 measure the transaction manager's initialization time while Test 3 is intended to measure the transaction execution time in a real world OLTP scenario. It can be executed in a single or a multiple transaction mode. In the single transaction mode we measure time per transaction without conflicts. On the other hand in the multiple transaction mode we measure transaction throughput. Finally, we can measure the transaction's execution time regarding to transaction manager. This mode has the following execution plan:

- 40 parallel transactions at a time
- each transaction is executing Test 3

- 5 execution repetitions.

The result is the amount of time spent on that execution.

5.4 Benchmarking environment

The environment used for executing the benchmark conforms to the Transaction Manager's implementation. The Transaction manager is implemented in Java and compiled into the byte-code and executed in the Java Virtual Machine (JVM). The JVM implementation has a significant influence on the Transaction Manager's performance, because the byte-code can be preprocessed and optimized during the test run. Hence, we can observe that the tested methods are executing faster during the test repetitions thanks to inline caches and JVM optimizations. The computer used for performing the tests was Intel Core 2 Duo, 2.0 GHz, HDD SATA 5400 r.p.m. with operating system Windows Vista 32-bit with Java Rutime Environment version 1.6.0_07.

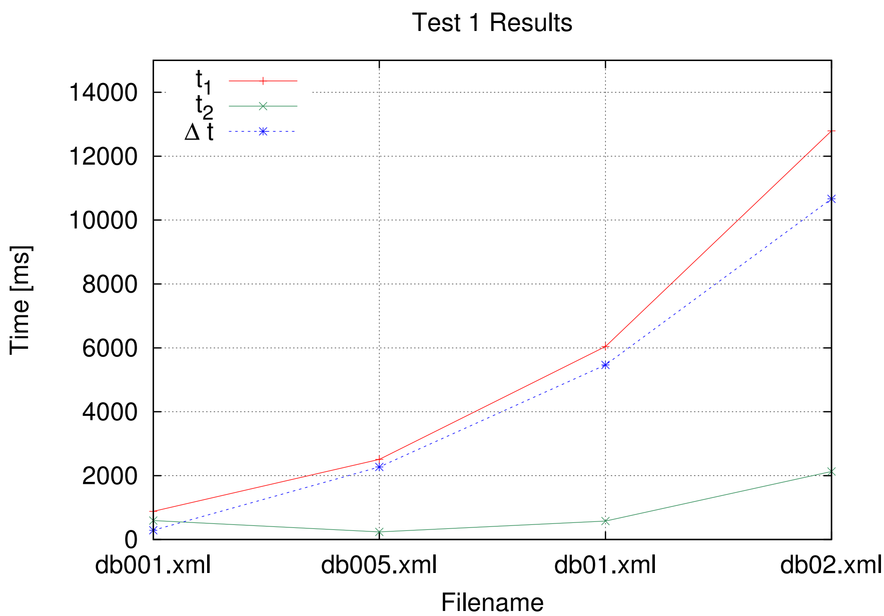


Figure 5.1: Test 1 results

Test 1	t_1 = document initialization with DeweyID ordering t_2 = document initialization without DeweyID ordering Result: $\Delta t = t_1 - t_2$
Test 2	t_1 = DOM operation getNode() with Transaction Manager t_2 = DOM operation getNode() without Transaction Manager Result: $\Delta t = t_1 - t_2$

Test 3	This test is intended to measure the transaction performance of the Transaction Manager's implementation. The schema S of the transaction consists of following operations. The semantics of these operations is described in Table 5.3.
	
	`BEGIN_TRANSACTION`
	`WAIT`
	`BID`
	`WAIT`
	`CLOSE_AUCTION`
	`WAIT`
	`INSERT_AUCTION`
	`WAIT`
	`GET_CATEGORIES`
	`WAIT`
	`REMOVE_ITEM`
	`COMMIT_TRANSACTION`
	
	t_1 = preceding schema S with Transaction Manager
	t_2 = preceding schema S without Transaction Manager
	Result: $\Delta t = t_1 - t_2$

Table 5.2: Description of tests

Operation	Semantics
WAIT	transaction waits a random time (0-?5000ms)
BID	bids on a random item in a random auction
CLOSE_AUCTION	moves random auction to closed auctions
INSERT_AUCTION	inserts new auction on a random item
REMOVE_ITEM	removes random item including all referenced auctions

Table 5.3: Semantics of transaction's operations

File Name	t_1 [ms]	t_2 [ms]	Δt [ms]
db001.xml	883	592	291
db005.xml	2510	239	2271
db01.xml	6042	577	5465
db02.xml	12794	2131	10663

Table 5.4: Test 1 results

File Name	t_1 [ms]	t_2 [ms]	Δt [ms]
db001.xml	182	131	51
db005.xml	217	84	133
db01.xml	325	121	204
db02.xml	380	154	226

Table 5.5: Test 2 results

File Name	t_1 [ms]	t_2 [ms]	Δt [ms]
db001.xml	3762	3928	-166
db005.xml	3784	3644	140
db01.xml	4444	4726	-282
db02.xml	10293	10402	-109

Table 5.6: Test 3 results - 20 transactions

5.4.1 Results

This section sums up our results, where each test was executed five times. At the beginning of each test there was an initialisation. This is important because JVM is loading classes when they are invoked for the first time.
The results of Test 1 are exposed in Table 5.4. In Graph 5.1 a linear dependency of Δt to

File Name	t_1 **[ms]**	t_2 **[ms]**	Δt **[ms]**
db001.xml	3739	4047	-308
db005.xml	5291	4626	665
db01.xml	9550	9483	67
db02.xml	29153	36494	-7341

Table 5.7: Test 3 results - 50 transactions

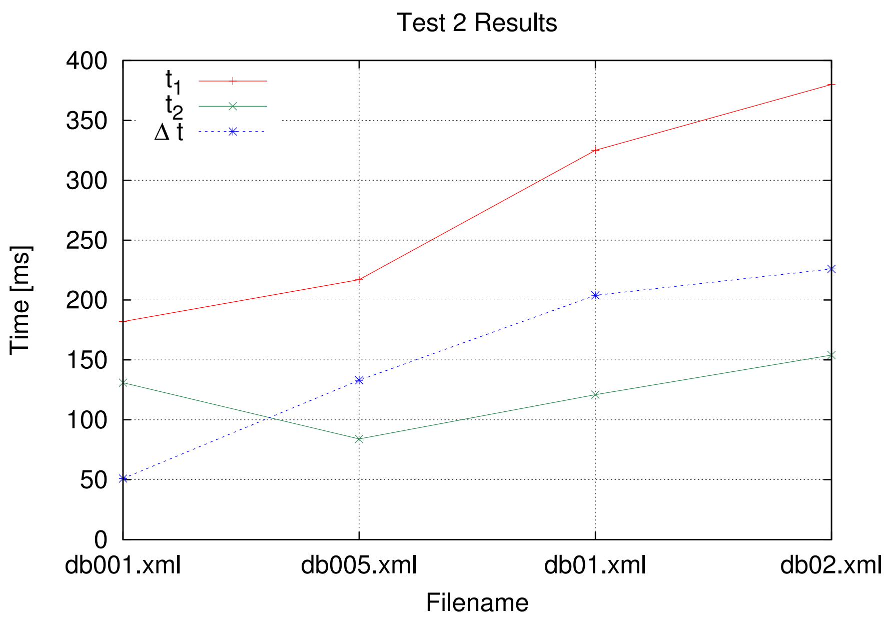


Figure 5.2: Test 2 results

the database instance is shown. The cost of the DeweyID ordering algorithm is approximately 90% of the time needed to build and initialize a database instance.

The results of Test 2 are displayed in Table 5.5. Relation of Δt to the database instance is depicted in Graph 5.2. This relation seems to be a sublinear function. This behavior is caused by the implementation of a DeweyID accessor that is implemented by a hash table. The time complexity of a search operation in a hash table is $O(1)$, a constant. But there is a small overhead of the Transaction Manager that has time complexity $O(n)$, hence the relation is not a constant.

We executed Test 3 in multiple transaction mode. It means that the benchmark executes transactions in a concurrent mode. The waiting time between nearby operations was 1000 ms. We did two measurements with different settings. The first setting included 20 concurrent transactions. The second one had 50 concurrent transactions. The measurement

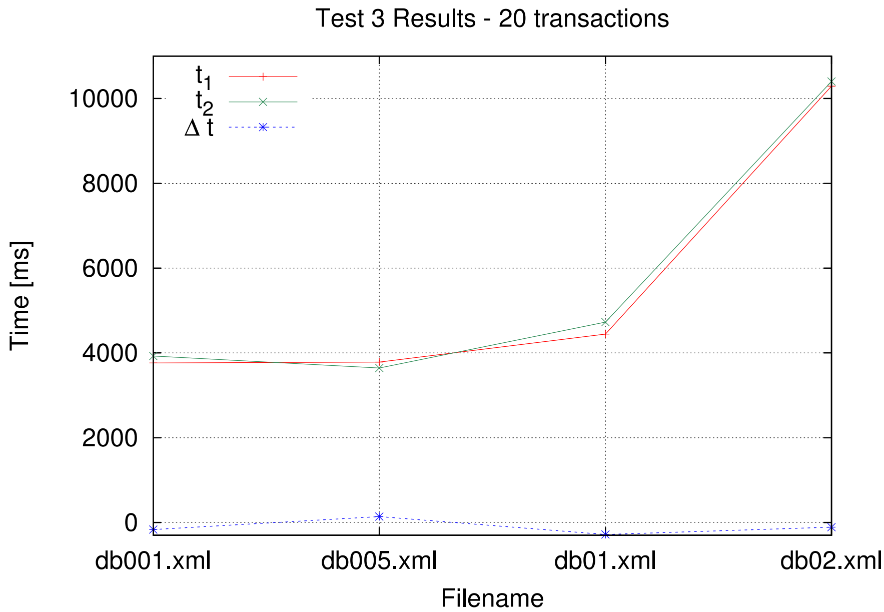


Figure 5.3: Test 3 results - 20 transactions

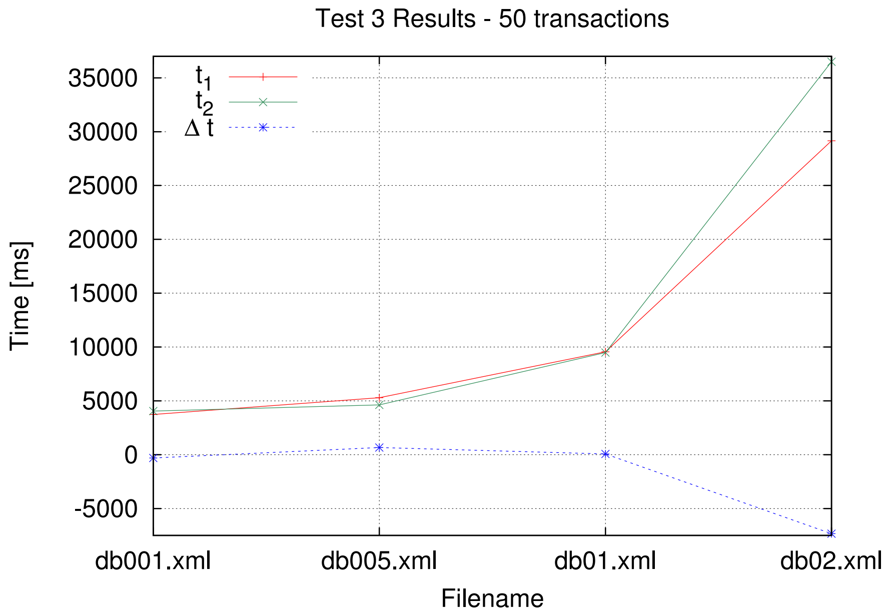


Figure 5.4: Test 3 results - 50 transactions

results of these settings are displayed in Tables 5.6 and 5.7. The corresponding graphs are in Figures 5.3 and 5.4. There is a significant result in Figure 5.4. The execution for 50 transactions is faster with Transaction Manager surprisingly. This effect is probably caused by inline caches of the Java Virtual Machine. The exception handling has an important impact on the performance. Many exceptions do not arise in the Transaction Manager environment because conflicting operations are suspended.

6 Prototypes

As a part of a work on this dissertation thesis we have implemented two prototypes of XML databases. The first prototype was written in Smalltalk and is called *CellStore*. Project *CellStore* represents native XML database and it was started in 2006 by Jan Vraný et al. We have used *CellStore* as a test-bed for testing and benchmarking implementation of transaction protocols mentioned earlier. The performance comparison of *CellStore's* XQuery Processor with other well-known processors is in Appendix C. The prototype is desribed in detail in [41]. In this chapter we give its brief description.

The second prototype is called RedXML and is implemented in Ruby. Its development and testing is a subject of our future work. The basic idea of the prototype is described in [59].

6.1 CellStore Native XML DBMS

The main goal of project *CellStore* [67] is to develop a NXDBMS for both educational and research purposes. It is meant rather as an experimental platform than an in-box and ready-to-use database engine. We planed such an engine because the students can easily look inside it, understand and create new components for this engine such as, e.g., a built-in XSLT engine, a query optimizer, an index engine, an event-condition-action (ECA) processing, etc.

According to this goal the development platform had been chosen. Especially:

- it should be easy to change of functionality of subsystems,
- it should be purely object-oriented for development and design,
- it must enable component reusing, test-driven development and trace & log facilities for both debugging and educational purposes.

In the end we selected Smalltalk/X as the development platform.

6.1.1 History

The project was started in 2004 with the first implementation of storage subsystem. Implementation of part of XQuery functionality (2007) was the next step. Then implementation of modules for simple-indexing, DML, transactional processing, cache management, web-based approach, remote client, and test setting and evaluation environment followed from 2007 to 2009.

In 2008 a significant change in the system architecture had been done. Jan Vraný included Perseus framework into *CellStore*'s architecture. It brought really illustrative code debugger based on event mechanism. But, on the other hand, it also requires partial redesign of several already done subsystems and slightly slows down *CellStore* efficiency.

6.1.2 CellStore's State of The Art

There are two stages in *CellStore* history – before and after Perseus incorporation. The first – pre-Perseus stage – provided several relatively well integrated modules. *CellStore* worked as an embedded DBMS with partial implementation of XQuery 1.0. It had a database console, a transaction management and a monitoring tool. A comprehensive description of *CellStore* at this stage was published in [49].

In 2008 several new modules and subsystems were under development (e.g. web and line clients, DML module, testing tool etc.). At the same time, Jan Vraný started with Perseus implementation [66]. His work implied the necessity of partial redesign of several already developed modules as well as modules just under development. The redesign process was successfully done on new XQuery interpreter, partially on transaction manager, and continues (within master theses) on modules for DML and indexing. Some modules (web and line clients and testing tools) were not affected, others (namely cache management module) were not redesigned yet.

6.1.3 System Architecture

CellStore's architecture is depicted in Figure 6.1. It can be approached through several interfaces at different levels of services. The lowest layer – low level storage – consists of several cooperating modules. Modules depicted in solid boxes are already implemented, whereas modules in dotted boxes are not ready yet.

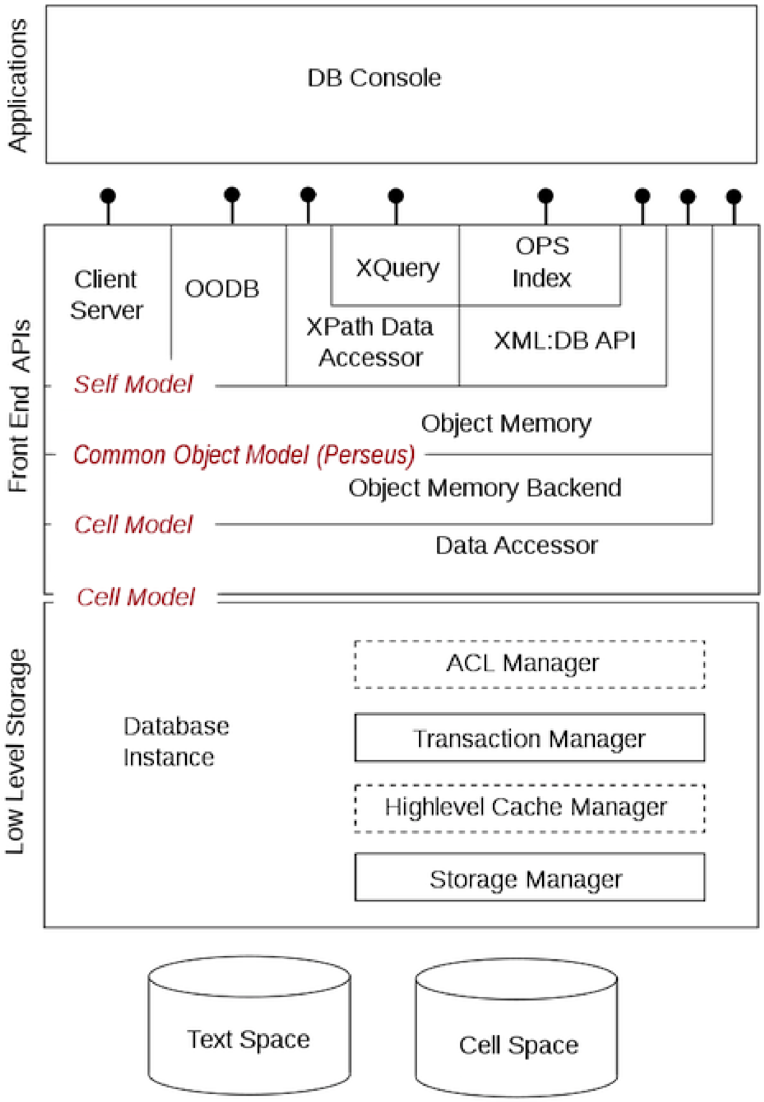


Figure 6.1: *CellStore* architecture

6.1.4 Storage Subsystem

We developed a new method for storing XML data. The method is based on work of [64] and partially inspired by solutions used in DBMSs Oracle[1] and Gemstone[2]. Structural and data parts of an XML document are stored separately. Of course, it increases necessary

[1] `http://www.oracle.com/us/products/database/index.html`
[2] `http://www.gemstone.com/products/gemstone`

time to store and reconstruct documents. But, on the other hand, it provides a great benefit in disk space management especially in case of document update, query processing and indexing of the stored XML data.

Let us describe the storage model in more detail. Note that the description is based on the first implementation version, because it is more illustrative. There exist improvements in the newer versions of *CellStore*, but they are not so important for a quick view. XML data documents are parsed and placed in two different files during the storing process – *cell file* and *data file*. We illustrate the structure of both the files using the following sample XML document:

```
<?xml version="1.0"?>
<!DOCTYPE simple PUBLIC
 "-//CVUT//Simple Example DTD 1.0//EN" SYSTEM simple.dtd">
<simple>
<!-- First comment -->
<?forsomeone process me?>
  <element xmlns="namespace1">
   First text
  <ns2:element xmlns:ns2="namespace2"
     attribute1="value1" ns2:attribute2="value2">
  </ns2:element>
  <empty/>
  </element>
</simple>
```

6.1.4.1 Cell File Structure

A *cell file* consists of fixed-length cells. Each *cell* represents a single DOM object (document, element, attribute, character data, etc.) or XML:DB API object (collection or resource). Note that this API is developed by XML:DB Initiative for XML Databases [69]. Cells are organized into fixed-length block.

A database *block* is the smallest I/O unit of transfer between disk and low-level storage cache. Only cells from one document can be stored in one block. The set of blocks describing the structure of the whole document is called a *segment*. Each block starts with header with a bitmap describing the density of the block.

Inside the cell structure internal pointers are used to represent parent-child and sibling

Name	**Content**	**Meaning**
Head	1 byte	The type of cell.
Parent	cell pointer	Pointer to parent cell.
Child	cell pointer	Pointer to the first child.
Sibling	cell pointer	Pointer to the next cell brother (NIL if there is no one).
D1, D2, D3, D4	depends on type	Contain either data or pointers (to a text file or a tag file) depending on the type of cell.

Table 6.1: *CellStore* cell structure

relationships of nodes. Each cell consists of eight fields, whereas their meaning can differ with different types of cells. The following cell types are supported in the system: character data, attribute, document, document type, processing instruction, comment, XML Resource, and collection. The general structure of cell is described in Table 6.1.

See Figure 6.2 to grasp the idea how the cell storage looks for the sample XML document mentioned above.

Cell Block #112233

	Head	Parent	Child	Sibling	D1	D2	D3	D4	
0x00	7F:F0:00:00	00:00:00:00	00:00:00:00	00:00:00:00	00:00:00:00	00:00:00:00	00:00:00:00	00:00:00:00	Free Cell Bitmap
0x01	09:00:00:00	010101:44	112233:03		112233:02				Document Cell
0x02	0A:00:00:00	112233:01			00000001	001122:01			<! DOCTYPE...
0x03	01:00:00:00	112233:01	112233:04		00000002				<simple>
0x04	08:00:00:00	112233:03		112233:05	112233:02				<!-- First comm
0x05	07:00:00:00	112233:03		112233:06	00000003	001122:03			<?forsomeone ...
0x06	01:00:00:00	112233:03	112233:07		00000004		00000005		<element ...
0x07	03:00:00:00	112233:06		112233:08	001122:04				First text
0x08	01:00:00:00	112233:06		112233:0B	00000004	00000007	00000006	112233:09	<ns2:element ...
0x09	02:00:00:00	112233:08		112233:0A	00000008		00000005	001122:05	attribute1="val...
0x0A	02:00:00:00	112233:08			00000009	00000007	00000006	001122:06	ns2:attribute2...
0x0B	01:00:00:00	112233:06			0000000A		00000005		<empty/>
0x0C									
0x0D									
0x0E									
0x0F									

Figure 6.2: *CellStore* cell file structure

6.1.4.2 Text File Structure

A *text file* contains all text data (i.e. contents of DOM text elements and attributes). The data is organized into *blocks* too, whereas one block belongs just to one document. The set of data blocks belonging to one document is called again a *segment*. A *text pointer* is a pointer to a text file. It consists of a *text block* and a *record*. Each text block contains a translation table which accepts a record number and returns the offset and the length of the data block. This strategy ensures efficiency in case of data changes. The translation table grows from the end of block, while data grow from the beginning. For these purposes the translation table contains the number of actual records. The header of a text block contains also a pointer to the root of its cell node necessary for full-text searching. A sample content of text file structure is shown on Figure 6.3.

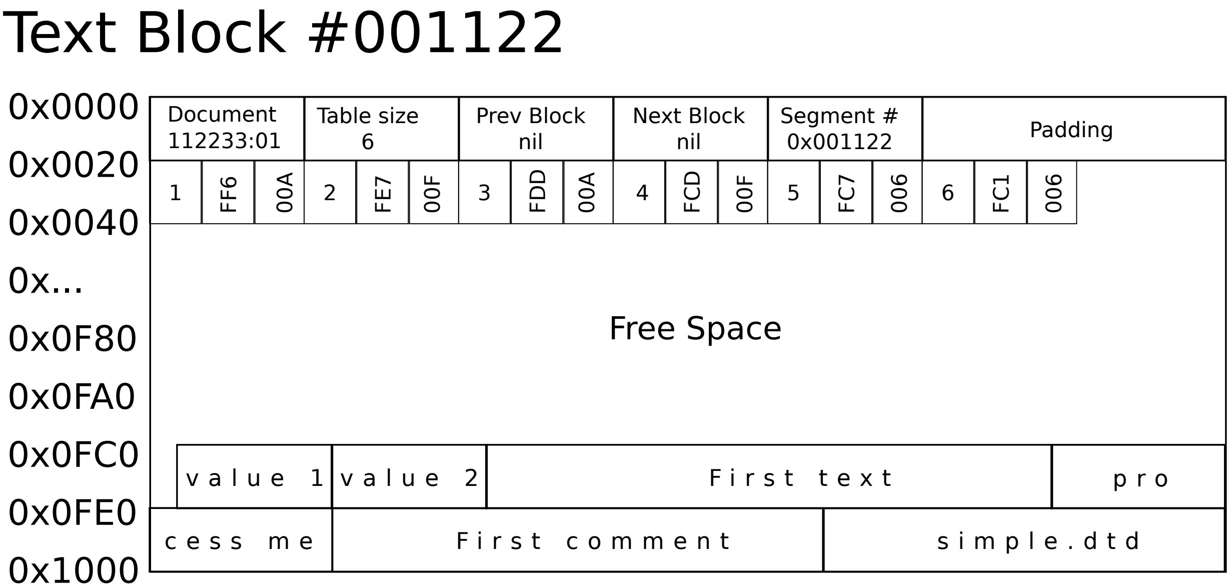


Figure 6.3: *CellStore* text file structure

The low-level subsystem was fully implemented and its stability was tested on INEX data set. INEX [24] is the set of articles from IEEE which contains approximately 12,000 individual XML documents (without figures) with total size of about 500MB.

The newer version of low-level subsystem implementation allows for individual setting of cell, cell-pointer, and block sizes. All these parameters can be used to optimize low-level storage according to specific data needs[3]. Unfortunately, we did not provide enough

[3]Similarly, in Oracle DBMS a `BLOCK_SIZE`, `PCT_FREE`, and extent-allocation parameters can be used to optimize storage.

experiments yet to be able to approve efficiency of such low-level customization.

6.1.4.3 The Transaction Manager Implementation

The CellStore's Transaction Manager consists of three independent modules - Transaction Manager, Lock Manager, and Log Manager. The most important one among them is the Lock Manager, which ensures the locking mechanism. The Transaction Manager is the encapsulation of the Lock Manager and Log Manager. The Log Manager is the helper class that encapsulates logging inside the Transaction Manager.

The UML class diagram of the Transaction Manager is shown in Figure 6.4.

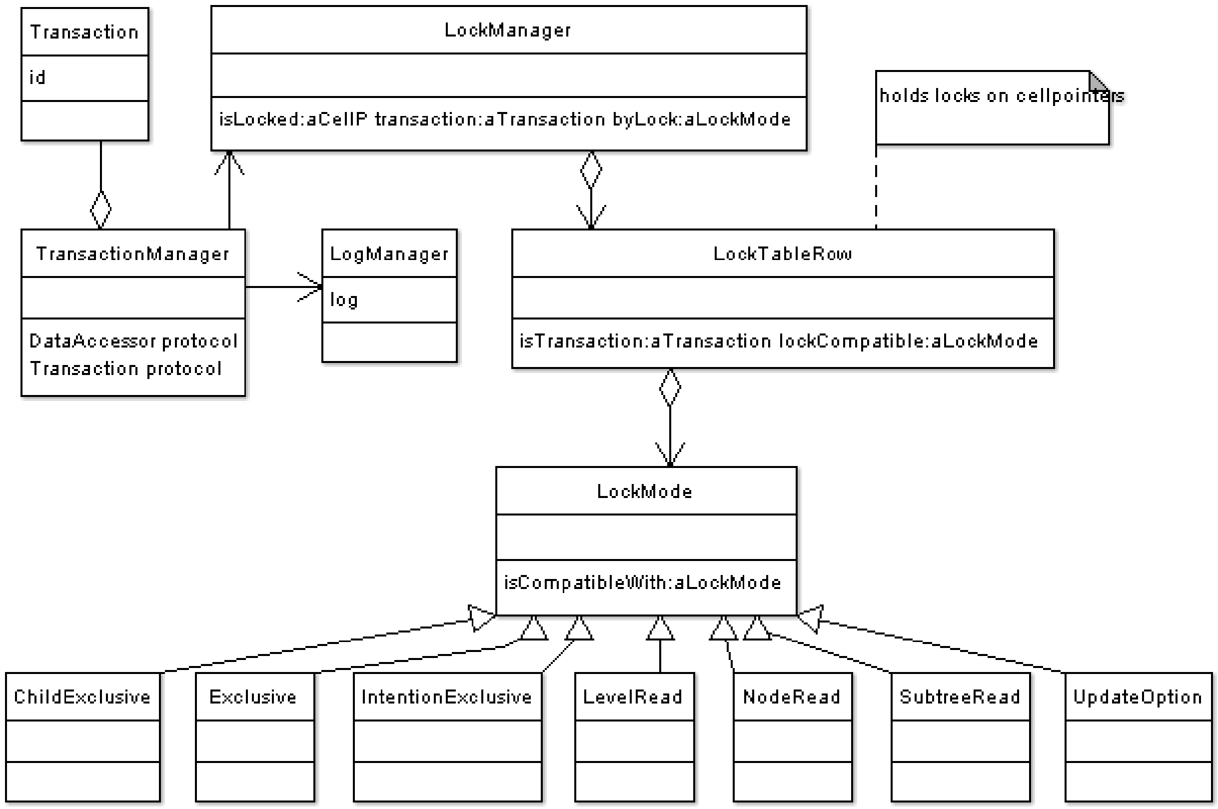


Figure 6.4: The Transaction Manager Class Diagram

The Transaction Manager provides methods for transaction processing as a *begin transaction*, *commit transaction* and *abort transaction*. In the CellStore project, the transaction manager is the transaction layer, which implements the DataAccessor protocol, because it have to be compatible with the other layers, especially with a cache layer, e.g. Cache Manager.

The Lock Manager is the necessary class for the Transaction Manager, because it implements the locking mechanism. As you can see in Figure 6.4 the locking mechanism is implemented by two methods:

```
LockManager>>isLocked:aCellP transaction:aTransaction
             byLock:aLockMode
LockTableRow>>isTransaction:aTransaction lockCompatible:aLockMode
```

The relationship between the LockManager and the LockTableRow is called the lockTable. The lockTable is implemented by a dictionary that is associated with the LockTableRow by aCellPointer.

6.1.4.4 Storage Discussion

Our storage strategy has an obvious drawback – necessity to divide XML data into text and structure parts during the storing process and their joining during the document reconstruction. On the other hand, it was experimentally shown, that the space requirement of our storage method is acceptable even in case of frequent changes of parts of stored data. Moreover, selected obvious improvements like using convenient compress algorithms for text space are evident, although they are not approved by experiments yet.

We believe that our storage method can also provide significant benefits in XQuery processing. Of course, it requires well designed and complex (XQuery) optimizer, which is able to guess and decide when to prefer text and when structure selection criteria. And, separation of structural and text information may also allow us to apply special indexing algorithms. However, all these notions are still at the level of hypothesis and future work.

7 Conclusions

In the first part of this thesis we have presented a formal XQuery Update Facility (XQUF) semantics extended by transaction processing features. To achieve this goal we had to extend XQUF syntax by new transaction specific syntax constructs covering needs of the transaction control. This was the main motivation for our work; to provide a complex framework to cope with transaction processing in (native) XML databases. The motivation for this work is our opinion that the XQUF semantics should provide built-in features for transaction processing. Hence, we decide to propose the alternative XQUF semantics implementing features supporting transaction processing.

To prove the correctness of the semantics' specification we implemented a prototype in Maude system [15] called XQUF-LP framework. In this framework we have covered basic constructs of the semantics, we claim that the finalization of the entire semantics' specification is only a technical issue which can be solved by master's students in further work. XQUF-LP framework can be used for proving algebraic features of the semantics.

Along with the theoretical part of the research we have designed and developed XPath, XQuery and XQuery Update Facility (including transaction processing) processors as modules in native XML database CellStore. We also designed a component benchmark to measure its performance. The results of the benchmark are promising (in the environment of the CellStore) but cannot compete with the state-of-the-art competitors [1].

7.1 Contributions

This thesis summarizes our research work in the past few years and provides the accomplished achievements during this period. We would like to emphasize the following contributions:

- We introduced detailed formal semantics of the XQuery Update Facility languge, a functional XML update language standardized by W3C, extended by transaction processing. This semantics can be used for verification of concurrent programs using this language, moreover the semantics is suitable to be the part of the standard in the future.

[1]The comparison can be found in [41].

- We extended XQuery Update Facility syntax and semantics by expressions for transaction control. These expressions are needed to control program flow according to transaction processing. This extension is suitable to be the part of the standard in the future. The main advantage of this approach lies in unified approach to transaction processing for future implementations.

- We provided a new simple benchmark for measuring overhead of a transaction manager module. This benchmark can be used for component based systems in the future.

- We specified a transaction processing for XML-λ Language by mapping its operations into DOM operations and utilizing taDOM locking protocol.

- We provided semantics verification by the prototype implementation in Maude system. This prototype implementation is very useful for formal proving of algebraic features of the language such as confluence or coherence. It can be used to verify new features added to the language in the future.

- We provided a prototype implementation of the native XML database CellStore, which is used as a testbed for experiments. This prototype is highly utilized by students for their seminar and (under-)graduate projects.

Last but not least, the contribution that should be certainly mentioned is the influence of the CellStore and transaction processing research on undergraduate and graduate students of our department. There were open many interesting topics for the semester and bachelor/masters projects aiming at improving the excellence both of the students and the research project. We believe that our efforts yielded benefits for all the participants.

7.2 Future Work

During writing the thesis we identified many open issues in our research. (1) The proposed semantics of the XQuery Update Facility language is not fully implemented in Maude system. We plan to do it in the near future. This implementation will be used to prove algebraic features of the locking protocol XQUF-LP. It can also be used to verify XQUF-LP's performance (number of operations) according to other proposed protocols mentioned in Chapter 3. (2) We investigate new possibilities for storing XML documents. We implemented a prototype called RedXML which is written in Ruby and utilizes Redis key-value database as a storage for XML documents. On the top of it we built an XQuery (Update Facility) processor with built-in transactional processing. The basic concept of the RedXML was presented in [59]. We plan to enhance and measure our new algorithms for mapping XML documents into a key-value store.

8 Bibliography

[1] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78. IEEE Computer Society, 2000.

[2] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling.*, volume II. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1973.

[3] R. Bača, J. Walder, M. Pawlas, and M. Krátký. Benchmarking the compression of XML node streams. In *Proceedings of the 15th international conference on Database systems for advanced applications*, DASFAA'10, pages 179–190, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, 1st edition, 1997.

[5] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation, 2010. `http://www.w3.org/TR/xquery/`.

[6] T. Böhme and E. Rahm. XMach-1: A benchmark for XML data management. In *Proceedings of BTW2001*, pages 264–273, 2001.

[7] T. Bray, F. Yergeau, J. Cowan, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.1, February 2004. http://www.w3.org/TR/2004/REC-xml11-20040204/.

[8] S. Bressan, M.-L. Lee, Y. G. Li, Z. Lacroix, and U. Nambiar. The XOO7 Benchmark. In *EEXTT*, pages 146–147, 2002.

[9] S. Bressan, Y. G. Li, G. Dobbie, Z. Lacroix, M. L. Lee, U. Nambiar, and B. Wadhwa. XOO7: applying OO7 benchmark to XML query processing tool. In *Proceedings of the tenth international conference on Information and knowledge management*, pages 167–174. ACM, 2001.

[10] C. Byun, I. Yun, and S. Park. A New Optimistic Concurrency Control in Valid XML. *J. Inf. Sci. Eng.*, 25(1):11–31, 2009.

[11] D. Chamberlin, A. Berglund, and S. Boag. XML Path Language (XPath) 2.0, December 2010. http://www.w3.org/TR/xpath20/.

[12] D. Chamberlin, D. Florescu, J. Robie, J. Melton, and J. Simon. XQuery Update Facility 1.0. http://www.w3.org/TR/xquery-update-10/.

[13] A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management - Native XML and XML-Enabled Database Systems*. Addison Wesley Professional, 2003. ISBN: 0-201-84452-4.

[14] T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 455–466, New York, NY, USA, 2005. ACM.

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[16] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[17] C. J. Date. *An Introduction to Database Systems, 6th Edition*. Addison-Wesley, 1995.

[18] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, September 2005. http://www.w3.org/TR/xquery-semantics/.

[19] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Recommendation, 2010.

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

[21] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM), December 2010. http://www.w3.org/TR/xpath-datamodel/.

[22] J. Gray and kolektiv. Granularity of Locks and Degrees of Consistency in a Shared Database. *Modeling in Data Base Management Systems*, pages 365–394, 1976.

[23] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann Publishers, 1st edition, 1993.

[24] N. Gvert and G. Kazai. Overview of the INitiative for the Evaluation of XML retrieval (INEX) 2002. In *In Fuhr et al*, pages 1–17. ERCIM, 2003.

[25] M. Haustein and T. Härder. Optimizing lock protocols for native XML processing. *Data Knowl. Eng.*, 65(1):147–173, 2008.

[26] M. Haustein and T. Harder. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. *In: Proc. ADBIS 2003, Dresden*, 1:88–102, Sept. 2003.

[27] M. P. Haustein and T. Härder. A synchronization concept for the DOM API. In H. Höpfner, G. Saake, and E. Schallehn, editors, *Grundlagen von Datenbanken*, pages 80–84. Fakultät für Informatik, Universität Magdeburg, 2003.

[28] M. P. Haustein and T. Härder. An efficient infrastructure for native transactional XML processing. *Data Knowl. Eng.*, 61(3):500–523, 2007.

[29] M. P. Haustein and T. Härder. XTC Project. `http://wwwlgis.informatik.uni-kl.de/cms/?id=36`, 2007.

[30] M. P. Haustein and T. Härder. taDOM scenarios. `http://wwwlgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/agdbis/projects/xtc/report/taDOM.html`, 2012.

[31] M. P. Haustein, T. Härder, C. Mathis, and M. W. 0002. DeweyIDs - The Key to Fine-Grained Management of XML Documents. In C. A. Heuser, editor, *SBBD*, pages 85–99. UFU, 2005.

[32] S. Helmer, C.-C. Kanne, and G. Moerkotte. Evaluating Lock-based Protocols for Cooperation on XML Documents. *SIGMOD RECORD*, 33:58–63, 2004.

[33] IBM. Comparison of the XML model and the relational model. `http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.apdv.embed.doc/doc/c0023811.htm`, May 2008.

[34] T. X. Initiative. XML:DB API Draft. `http://xmldb-org.sourceforge.net/xapi/xapi-draft.html`, 2001.

[35] ISO. *ISO/IEC 9075-1:2011 Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework).* Dec. 2011.

[36] K.-F. Jea and S.-Y. Chen. A high concurrency XPath-based locking protocol for XML databases. *Information and Software Technology*, 48(8):708 – 716, 2006.

[37] P. Jíra, M. Kostolný, and P. Strnad. RedXML Project. `https://github.com/jirapave/RedisXmlConcept`, 2013.

[38] S. C. Kleene. Representation of events in nerve nets and finite automata. 1951.

[39] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[40] P. Loupal. *XML-λ : A functional framework for XML.* PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, February 2010.

[41] P. Loupal, I. Mlykova, M. Necasky, K. Richta, and P. Strnad. Storing XML Data - The ExDB and CellStore Way in the Context of Current Approaches. *INFORMATICA*, 23(2):247–282, 2012.

[42] N. Marti-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In *Handbook of Philosophical Logic*, pages 1–87. Springer, 2002.

[43] M. Nicola, I. Kogan, R. Raghu, A. Gonzalez, B. Schiefer, and K. Xie. Transaction Processing over XML (TPoX). `http://tpox.sourceforge.net/TPoX_BenchmarkProposal_v1.2.pdf`, 2008.

[44] M. Nicola, I. Kogan, and B. Schiefer. An XML transaction processing benchmark. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 937–948, New York, NY, USA, 2007. ACM Press.

[45] P. Pleshachkov, P. Chardin, and S. Kuznetsov. A DataGuide-Based Concurrency Control Protocol for Cooperation on XML Data. In J. Eder, H.-M. Haav, A. Kalja, and J. Penjam, editors, *Advances in Databases and Information Systems*, volume 3631 of *Lecture Notes in Computer Science*, pages 268–282. Springer Berlin Heidelberg, 2005.

[46] P. Pleshachkov, P. Chardin, and S. Kuznetsov. XDGL: XPath-Based Concurrency Control Protocol for XML Data. In M. Jackson, D. Nelson, and S. Stirk, editors, *Database: Enterprise, Skills and Innovation*, volume 3567 of *Lecture Notes in Computer Science*, pages 73–73. Springer Berlin / Heidelberg, 2005.

[47] P. Pleshachkov and S. Kuznetsov. SXDGL: Snapshot Based Concurrency Control Protocol for XML Data. In *XSym'07*, pages 122–136, 2007.

[48] P. Pleshachkov and L. Novak. Transaction Isolation in the Sedna Native XML DBMS. *SYRCoDIS 2004*, 2004.

[49] J. Pokorny, K. Richta, and M. Valenta. CellStore: Educational and Experimental XML-Native DBMS. *Information Systems Development: Challenges in Practice, Theory, and Education*, page 989, 2008.

[50] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[51] S. Sanfilippo. Redis Project. `http://redis.io`, 2013.

[52] R. Savage and J. Leffler. BNF Grammar for ISO/IEC 9075:1999 - Database Language SQL (SQL-99). `http://savage.net.au/SQL/sql-99.bnf.html`2008-05-20, 2004.

[53] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 974–985. VLDB Endowment, 2002.

[54] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.

[55] A. Siirtola and M. Valenta. Verifying parameterized taDOM+ lock managers. *SOFSEM 2008*, pages 460–472, 2008.

[56] P. Strnad. Rozvrhovač transakcí v projektu Cellstore (In Czech), 2007.

[57] P. Strnad. XQUF-LP Prototype Implementation. `https://github.com/strny/xquf-lp`, 2013.

[58] P. Strnad and P. Loupal. Using taDOM Locking Protocol in a Functional XML Update Language. *In Proc. DATESO 2008 Workshop*, 1:25–37, 2008.

[59] P. Strnad, O. Macek, and P. Jira. Mapping XML to Key-Value Database. In *DBKDA 2013, The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 121–127, 2013.

[60] P. Strnad and M. Valenta. Object-oriented Implementation of Transaction Manager in CellStore Project. *Objekty 2006, Praha*, pages 273–283, 2006.

[61] P. Strnad and M. Valenta. On Benchmarking of Transaction Managers. *In Database Systems for Advanced Applications, DASFAA 2009 International Workshops: BenchmarX, MBC, MCIS, PPDA, WDPP, PhD.*, 1, 2009.

[62] The W3C Consortium. Document Object Model (DOM), 2005. `http://www.w3.org/DOM/`.

[63] The W3C Consortium. W3C Homepage. `http://www.w3.org`, 2013.

[64] K. Toman. Storing XML Data In a Native Repository. In V. Snášel, J. Pokorný, and K. Richta, editors, *Proceedings of the Dateso 2004 Annual International Workshop on DAtabases, TExts, Specifications and Objects, Desna, Czech Republic, April 14-16, 2004*, volume 98 of *CEUR Workshop Proceedings*, pages 51–62. CEUR-WS.org, 2004.

[65] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. *Electronic Notes in Theoretical Computer Science*, 71:282–300, 2004.

[66] J. Vraný and A. Bergel. Perseus Framework. `http://swing.fit.cvut.cz/projects/perseus`, 2008.

[67] J. Vraný, P. Strnad, and M. Valenta. XML:DB API Draft CellStore Implementation. `https://swing.fit.cvut.cz/projects/cellstore`, 2010.

[68] J. Vraný, M. Valenta, and P. Strnad. CellStore Project. `https://swing.fit.cvut.cz/projects/cellstore/`, 2007.

[69] XMLDB. XML:DB API. `http://www.xmldb.org`, 2001.

9 Refereed publications of the author

[A.1] Strnad P., Macek O., Jira P. Mapping XML to Key-Value Database. *In DBKDA 2013, The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications*, Red Hook: Curran Associates, Inc., 2013, vol. 1, pages 121–127, 2013.

[A.2] Loupal P., Mlynkova I., Necasky M., Richta K., Strnad P. Storing XML Data - The ExDB and CellStore Way in the Context of Current Approaches. *Informatica*, vol. 23, no. 2, 247–282, 2012.

[A.3] Loupal P., Kantor A., Macek O., Strnad P. On Indexing in Native XML Database Systems. *In DATESO 2012*, Prague: MATFYSPRESS, 2012, pages 127–134, 2012.

[A.4] Strnad P., Valenta M. On Benchmarking Transaction Managers. *In Database Systems for Advanced Applications DASFAA 2009 International Workshops: BenchmarX, MCIS, WDPP, PPDA, MBC, PhD, Brisbane, Australia, April 20. - 23.*, Berlin: Springer, 2009, pages 79–92, 2009.

[A.5] Strnad P. Measurement of Transaction Throughput in Native XML Database. *In CTU Workshop 2009*, Prague: CTU, 2009, pages 108–109, 2009.

[A.6] Strnad P., Loupal P. Using taDOM Locking Protocol in a Functional XML Update Language. *In Proceedings of the Dateso 2008 Workshop*, Ostrava: Technical University of Ostrava, 2008, pages 25–37, 2008.

[A.7] Strnad P., Valenta M. Object-oriented Implementation of Transaction Manager in CellStore. *In Objekty 2006*, Ostrava: Technical University of Ostrava, 2006, pages 273–282, 2006.

A XQuery 1.0 Grammar with Updates and TCL

This Appendix contains XQuery Update Facility 1.0 EBNF grammar extended by transaction control expressions.

```
[1]     Module  ::=  VersionDecl? (LibraryModule | MainModule)
[2]     VersionDecl  ::=  "xquery" "version" StringLiteral
                          ("encoding" StringLiteral)? Separator
[3]     MainModule  ::=  Prolog QueryBody
[4]     LibraryModule  ::=  ModuleDecl Prolog
[5]     ModuleDecl  ::=  "module" "namespace" NCName "=" URILiteral
                         Separator
[6]     Prolog  ::=  ((DefaultNamespaceDecl
                       | Setter
                       | NamespaceDecl
                       | Import) Separator)*
                     ((VarDecl | FunctionDecl | OptionDecl) Separator)*
[7]     Setter  ::=  BoundarySpaceDecl
                     | DefaultCollationDecl
                     | BaseURIDecl
                     | ConstructionDecl
                     | OrderingModeDecl
                     | EmptyOrderDecl
                     | RevalidationDecl
                     | CopyNamespacesDecl
                     | TransactionDecl
[8]     Import  ::=  SchemaImport | ModuleImport
[9]     Separator  ::=  ";"
[10]    NamespaceDecl  ::=  "declare" "namespace" NCName "=" URILiteral
[11]    BoundarySpaceDecl  ::=  "declare" "boundary-space"
                                ("preserve" | "strip")
[12]    DefaultNamespaceDecl  ::=  "declare" "default"
                                   ("element" | "function")
                                   "namespace" URILiteral
[13]    OptionDecl  ::=  "declare" "option" QName StringLiteral
```

```
[14]  OrderingModeDecl ::= "declare" "ordering"
                            ("ordered" | "unordered")
[15]  EmptyOrderDecl ::= "declare" "default" "order"
                          "empty" ("greatest" | "least")
[16]  CopyNamespacesDecl ::= "declare" "copy-namespaces"
                              PreserveMode "," InheritMode
[17]  PreserveMode ::= "preserve" | "no-preserve"
[18]  InheritMode ::= "inherit" | "no-inherit"
[19]  DefaultCollationDecl ::= "declare" "default" "collation"
                                URILiteral
[20]  BaseURIDecl ::= "declare" "base-uri" URILiteral
[21]  SchemaImport ::= "import" "schema" SchemaPrefix? URILiteral
                        ("at" URILiteral ("," URILiteral)*)?
[22]  SchemaPrefix ::= ("namespace" NCName "=")
                        | ("default" "element" "namespace")
[23]  ModuleImport ::= "import" "module" ("namespace" NCName "=")?
                        URILiteral ("at" URILiteral ("," URILiteral)*)?
[24]  VarDecl ::= "declare" "variable" "$" QName TypeDeclaration?
                   ((":=" ExprSingle) | "external")
[25]  ConstructionDecl ::= "declare" "construction"
                            ("strip" | "preserve")
[26]  FunctionDecl ::= "declare" "updating"? "function" QName
                        "(" ParamList? ")"
                        ("as" SequenceType)? (EnclosedExpr | "external")
[27]  ParamList ::= Param ("," Param)*
[28]  Param ::= "$" QName TypeDeclaration?
[29]  EnclosedExpr ::= "{" Expr "}"
[30]  QueryBody ::= Expr
[31]  Expr ::= ExprSingle ("," ExprSingle)*
[32]  ExprSingle ::= FLWORExpr
                  | QuantifiedExpr
                  | TypeswitchExpr
                  | IfExpr
                  | InsertExpr
```

```
                        | DeleteExpr
                        | RenameExpr
                        | ReplaceExpr
                        | TransformExpr
                        | OrExpr
                        | BeginExpr
                        | CommitExpr
                        | RollbackExpr
[33]    FLWORExpr  ::=  (ForClause | LetClause)+ WhereClause?
                        OrderByClause? "return" ExprSingle
[34]    ForClause  ::=  "for" "$" VarName TypeDeclaration?
                        PositionalVar? "in"
                        ExprSingle ("," "$" VarName TypeDeclaration?
                                    PositionalVar? "in" ExprSingle)*
[35]    PositionalVar  ::=  "at" "$" VarName
[36]    LetClause  ::= "let" "$" VarName TypeDeclaration? ":=" ExprSingle
                       ("," "$" VarName TypeDeclaration? ":=" ExprSingle)*
[37]    WhereClause  ::=  "where" ExprSingle
[38]    OrderByClause  ::=  (("order" "by") | ("stable" "order" "by"))
                            OrderSpecList
[39]    OrderSpecList  ::=  OrderSpec ("," OrderSpec)*
[40]    OrderSpec  ::=  ExprSingle OrderModifier
[41]    OrderModifier  ::=  ("ascending" | "descending")?
                            ("empty" ("greatest" | "least"))?
                            ("collation" URILiteral)?
[42]    QuantifiedExpr  ::=  ("some" | "every") "$" VarName
                             TypeDeclaration? "in" ExprSingle
                             ("," "$" VarName TypeDeclaration?
                              "in" ExprSingle)* "satisfies" ExprSingle
[43]    TypeswitchExpr  ::=  "typeswitch" "(" Expr ")"
                             CaseClause+ "default" ("$" VarName)?
                             "return" ExprSingle
[44]    CaseClause  ::=  "case" ("$" VarName "as")? SequenceType
                         "return" ExprSingle
```

```
[45]  IfExpr  ::=  "if" "(" Expr ")" "then"
                   ExprSingle "else" ExprSingle
[46]  OrExpr  ::=  AndExpr ( "or" AndExpr )*
[47]  AndExpr  ::=  ComparisonExpr ( "and" ComparisonExpr )*
[48]  ComparisonExpr  ::=  RangeExpr ( (ValueComp
                           | GeneralComp
                           | NodeComp) RangeExpr )?
[49]  RangeExpr  ::=  AdditiveExpr ( "to" AdditiveExpr )?
[50]  AdditiveExpr  ::=  MultiplicativeExpr
                         ( ("+" | "-") MultiplicativeExpr )*
[51]  MultiplicativeExpr  ::=  UnionExpr ( ("*"
                                          | "div"
                                          | "idiv"
                                          | "mod") UnionExpr )*
[52]  UnionExpr  ::=  IntersectExceptExpr
                      ( ("union" | "|") IntersectExceptExpr )*
[53]  IntersectExceptExpr  ::=  InstanceofExpr ( ("intersect"
                                                 | "except")
                                                InstanceofExpr )*
[54]  InstanceofExpr  ::=  TreatExpr ( "instance" "of" SequenceType )?
[55]  TreatExpr  ::=  CastableExpr ( "treat" "as" SequenceType )?
[56]  CastableExpr  ::=  CastExpr ( "castable" "as" SingleType )?
[57]  CastExpr  ::=  UnaryExpr ( "cast" "as" SingleType )?
[58]  UnaryExpr  ::=  ("-" | "+")* ValueExpr
[59]  ValueExpr  ::=  ValidateExpr | PathExpr | ExtensionExpr
[60]  GeneralComp  ::=  "=" | "!=" | "<" | "<=" | ">" | ">="
[61]  ValueComp  ::=  "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[62]  NodeComp  ::=  "is" | "<<" | ">>"
[63]  ValidateExpr  ::=  "validate" ValidationMode? "{" Expr "}"
[64]  ValidationMode  ::=  "lax" | "strict"
[65]  ExtensionExpr  ::=  Pragma+ "{" Expr? "}"
[66]  Pragma  ::=  "(#" S? QName (S PragmaContents)? "#)"
[67]  PragmaContents  ::=  (Char* - (Char* '#)' Char*))
[68]  PathExpr  ::=  ("/" RelativePathExpr?)
```

```
                  | ("//" RelativePathExpr)
                  | RelativePathExpr
[69]    RelativePathExpr  ::=  StepExpr (("/" | "//") StepExpr)*
[70]    StepExpr  ::=  FilterExpr | AxisStep
[71]    AxisStep  ::=  (ReverseStep | ForwardStep) PredicateList
[72]    ForwardStep  ::=  (ForwardAxis NodeTest) | AbbrevForwardStep
[73]    ForwardAxis  ::=  ("child" "::")
                     | ("descendant" "::")
                     | ("attribute" "::")
                     | ("self" "::")
                     | ("descendant-or-self" "::")
                     | ("following-sibling" "::")
                     | ("following" "::")
[74]    AbbrevForwardStep  ::=  "@"? NodeTest
[75]    ReverseStep  ::=  (ReverseAxis NodeTest) | AbbrevReverseStep
[76]    ReverseAxis  ::=  ("parent" "::")
                     | ("ancestor" "::")
                     | ("preceding-sibling" "::")
                     | ("preceding" "::")
                     | ("ancestor-or-self" "::")
[77]    AbbrevReverseStep  ::=  ".."
[78]    NodeTest  ::=  KindTest | NameTest
[79]    NameTest  ::=  QName | Wildcard
[80]    Wildcard  ::=  "*"
                  | (NCName ":" "*")
                  | ("*" ":" NCName)
[81]    FilterExpr  ::=  PrimaryExpr PredicateList
[82]    PredicateList  ::=  Predicate*
[83]    Predicate  ::=  "[" Expr "]"
[84]    PrimaryExpr  ::=  Literal
                     | VarRef
                     | ParenthesizedExpr
                     | ContextItemExpr
                     | FunctionCall
```

```
                          | OrderedExpr
                          | UnorderedExpr
                          | Constructor
[85]    Literal  ::=  NumericLiteral | StringLiteral
[86]    NumericLiteral  ::=  IntegerLiteral | DecimalLiteral
                             | DoubleLiteral
[87]    VarRef  ::=  "$" VarName
[88]    VarName  ::=  QName
[89]    ParenthesizedExpr  ::=  "(" Expr? ")"
[90]    ContextItemExpr  ::=  "."
[91]    OrderedExpr  ::=  "ordered" "{" Expr "}"
[92]    UnorderedExpr  ::=  "unordered" "{" Expr "}"
[93]    FunctionCall  ::=  QName "(" (ExprSingle ("," ExprSingle)*)? ")"
[94]    Constructor  ::=  DirectConstructor
                          | ComputedConstructor
[95]    DirectConstructor  ::=  DirElemConstructor
                                | DirCommentConstructor
                                | DirPIConstructor
[96]    DirElemConstructor  ::=  "<" QName DirAttributeList
                                 ("/>" | (">" DirElemContent*
                                          "</" QName S? ">"))
[97]    DirAttributeList  ::=  (S (QName S? "=" S? DirAttributeValue)?)*
[98]    DirAttributeValue  ::=  ('"' (EscapeQuot
                                     | QuotAttrValueContent)* '"')
                                | ("'" (EscapeApos
                                       | AposAttrValueContent)* "'")
[99]    QuotAttrValueContent  ::=  QuotAttrContentChar
                                   | CommonContent
[100]   AposAttrValueContent  ::=  AposAttrContentChar
                                   | CommonContent
[101]   DirElemContent  ::=  DirectConstructor
                             | CDataSection
                             | CommonContent
                             | ElementContentChar
```

```
[102]  CommonContent  ::=  PredefinedEntityRef
                           | CharRef | "{{" | "}}" | EnclosedExpr
[103]  DirCommentConstructor  ::=  "<!--" DirCommentContents "-->"
[104]  DirCommentContents  ::=  ((Char - '-') | ('-' (Char - '-')))*
[105]  DirPIConstructor  ::=  "<?" PITarget (S DirPIContents)? "?>"
[106]  DirPIContents  ::=  (Char* - (Char* '?>' Char*))
[107]  CDataSection  ::=  "<![CDATA[" CDataSectionContents "]]>"
[108]  CDataSectionContents  ::=  (Char* - (Char* ']]>' Char*))
[109]  ComputedConstructor  ::=  CompDocConstructor
                                 | CompElemConstructor
                                 | CompAttrConstructor
                                 | CompTextConstructor
                                 | CompCommentConstructor
                                 | CompPIConstructor
[110]  CompDocConstructor  ::=  "document" "{" Expr "}"
[111]  CompElemConstructor  ::=  "element" (QName | ("{" Expr "}"))
                                 "{" ContentExpr? "}"
[112]  ContentExpr  ::=  Expr
[113]  CompAttrConstructor  ::=  "attribute" (QName | ("{" Expr "}"))
                                 "{" Expr? "}"
[114]  CompTextConstructor  ::=  "text" "{" Expr "}"
[115]  CompCommentConstructor  ::=  "comment" "{" Expr "}"
[116]  CompPIConstructor  ::=  "processing-instruction"
                               (NCName | ("{" Expr "}")) "{" Expr? "}"
[117]  SingleType  ::=  AtomicType "?"?
[118]  TypeDeclaration  ::=  "as" SequenceType
[119]  SequenceType  ::=  ("empty-sequence" "(" ")")
                          | (ItemType OccurrenceIndicator?)
[120]  OccurrenceIndicator  ::=  "?" | "*" | "+"
[121]  ItemType  ::=  KindTest | ("item" "(" ")") | AtomicType
[122]  AtomicType  ::=  QName
[123]  KindTest  ::=  DocumentTest
                      | ElementTest
                      | AttributeTest
```

```
                     | SchemaElementTest
                     | SchemaAttributeTest
                     | PITest
                     | CommentTest
                     | TextTest
                     | AnyKindTest
[124]   AnyKindTest  ::=  "node" "(" ")"
[125]   DocumentTest  ::=  "document-node" "("
                           (ElementTest | SchemaElementTest)? ")"
[126]   TextTest  ::=  "text" "(" ")"
[127]   CommentTest  ::=  "comment" "(" ")"
[128]   PITest  ::=  "processing-instruction"
                    "(" (NCName | StringLiteral)? ")"
[129]   AttributeTest  ::=  "attribute" "("
                            (AttribNameOrWildcard ("," TypeName)?)? ")"
[130]   AttribNameOrWildcard  ::=  AttributeName | "*"
[131]   SchemaAttributeTest  ::=  "schema-attribute"
                                  "(" AttributeDeclaration ")"
[132]   AttributeDeclaration  ::=  AttributeName
[133]   ElementTest  ::=  "element" "(" (ElementNameOrWildcard
                                         ("," TypeName "?"?)?)? ")"
[134]   ElementNameOrWildcard  ::=  ElementName | "*"
[135]   SchemaElementTest  ::=  "schema-element" "(" ElementDeclaration ")"
[136]   ElementDeclaration  ::=  ElementName
[137]   AttributeName  ::=  QName
[138]   ElementName  ::=  QName
[139]   TypeName  ::=  QName
[140]   URILiteral  ::=  StringLiteral
[141]   RevalidationDecl  ::=  "declare" "revalidation"
                               ("strict" | "lax" | "skip")
[142]   InsertExprTargetChoice  ::=  (("as" ("first" | "last"))? "into")
                                     | "after"
                                     | "before"
[143]   InsertExpr  ::=  "insert" ("node" | "nodes") SourceExpr
```

```
                     InsertExprTargetChoice TargetExpr
[144]   DeleteExpr  ::=  "delete" ("node" | "nodes") TargetExpr
[145]   ReplaceExpr  ::=  "replace" ("value" "of")? "node"
                      TargetExpr "with" ExprSingle
[146]   RenameExpr  ::=  "rename" "node" TargetExpr "as" NewNameExpr
[147]   SourceExpr  ::=  ExprSingle
[148]   TargetExpr  ::=  ExprSingle
[149]   NewNameExpr  ::=  ExprSingle
[150]   TransformExpr  ::=  "copy" "$" VarName ":=" ExprSingle
                        ("," "$" VarName ":=" ExprSingle)*
                        "modify" ExprSingle "return" ExprSingle
[200]   BeginExpr ::= "BEGIN"
[201]   CommitExpr ::= "COMMIT"
[202]   RollbackExpr ::= "ROLLBACK"
[203]   TransactionDecl ::= "declare" "transaction" "isolation" "level"
                         (("read" ("uncommitted" | "committed"))
                         |("repeatable" "read")
                         |("serializable")
```

B Light-Weight XDM Specification

B.1 Model Elements

B.1.1 Document Nodes

Document Nodes encapsulate XML documents. Documents have the following properties:

- base-uri, possibly empty.
- children, possibly empty.
- unparsed-entities, possibly empty.
- document-uri, possibly empty.
- string-value
- typed-value

Document Nodes must satisfy the following constraints.

- The children must consist exclusively of Element and Text Nodes if it is not empty. Document Nodes can never appear as children.
- If a node N is among the children of a Document Node D, then the parent of N must be D.
- If a node N has a parent Document Node D, then N must be among the children of D.
- The string-value property of a Document Node must be the concatenation of the string-values of all its Text Node descendants in document order or, if the document has no such descendants, the zero-length string.

Accessors

dm:attributes

Returns the empty sequence

dm:base-uri

Returns the value of the base-uri property.

dm:children

Returns the value of the children property.

dm:document-uri

Returns the absolute URI of the resource from which the Document Node was constructed, or the empty sequence if no such absolute URI is available.

dm:is-id

Returns the empty sequence.

dm:is-idrefs

Returns the empty sequence.

dm:nilled

Returns the empty sequence.

dm:node-kind

Returns document.

dm:node-name

Returns the empty sequence.

dm:parent

Returns the empty sequence.

dm:string-value

Returns the value of the string-value property.

dm:type-name

Returns the empty sequence.

dm:typed-value

Returns the value of the typed-value property.

dm:unparsed-entity-public-id

Returns the public identifier of the specified unparsed entity or the empty sequence if no such entity exists.

dm:unparsed-entity-system-id

Returns the system identifier of the specified unparsed entity or the empty sequence if no such entity exists.

B.1.2 Element Nodes

Element Nodes encapsulate XML elements. Elements have the following properties:

- base-uri, possibly empty.
- node-name
- parent, possibly empty
- type-name
- children, possibly empty
- attributes, possibly empty
- nilled
- string-value
- typed-value
- is-id
- is-idrefs

Element Nodes must satisfy the following constraints.

- The children must consist exclusively of Element and Text Nodes if it is not empty.
- The Attribute Nodes of an element must have distinct xs:QNames.
- If a node N is among the children of an element E, then the parent of N must be E.

- Exclusive of Attribute Nodes, if a node N has a parent element E, then N must be among the children of E. (Attribute Nodes have a parent, but they do not appear among the children of their parent.)

- The data model permits Element Nodes without parents (to represent partial results during expression processing, for example). Such Element Nodes must not appear among the children of any other node.

- If an Attribute Node A is among the attributes of an element E, then the parent of A must be E.

- If an Attribute Node A has a parent element E, then A must be among the attributes of E.

- The data model permits Attribute Nodes without parents. Such Attribute Nodes must not appear among the attributes of any Element Node.

- If the dm:type-name of an Element Node is xs:untyped, then the dm:type-name of all its descendant elements must also be xs:untyped and the dm:type-name of all its Attribute Nodes must be xs:untypedAtomic.

- If the dm:type-name of an Element Node is xs:untyped, then the nilled property must be false.

- If the nilled property is true, then the children property must not contain Element Nodes or Text Nodes.

- For every expanded QName that appears in the dm:node-name of the element, the dm:node-name of any Attribute Node among the attributes of the element, or in any value of type xs:QName or xs:NOTATION (or any type derived from those types) that appears in the typed-value of the element or the typed-value of any of its attributes, if the expanded QName has a non-empty URI, then there must be a prefix binding for this URI among the namespaces of this Element Node.

- If any of the expanded QNames has an empty URI, then there must not be any binding among the namespaces of this Element Node which binds the empty prefix to a URI.

- The string-value property of an Element Node must be the concatenation of the string-values of all its Text Node descendants in document order or, if the element has no such descendants, the zero-length string.

Accessors

dm:attributes

Returns the value of the attributes property. The order of Attribute Nodes is stable but implementation dependent.

dm:base-uri

Returns the value of the base-uri property.

dm:children

Returns the value of the children property.

dm:document-uri

Returns the empty sequence.

dm:is-id

Returns the value of the is-id property.

dm:is-idrefs

Returns the value of the is-idrefs property.

dm:nilled

Returns the value of the nilled property.

dm:node-kind

Returns element.

dm:node-name

Returns the value of the node-name property.

dm:parent

Returns the value of the parent property.

dm:string-value

Returns the value of the string-value property.

dm:type-name

Returns the value of the type-name property.

dm:typed-value

Returns the value of the typed-value property.

dm:unparsed-entity-public-id

Returns the empty sequence.

dm:unparsed-entity-system-id

Returns the empty sequence.

B.1.3 Attribute Nodes

Attribute Nodes represent XML attributes. Attributes have the following properties:

- node-name
- parent, possibly empty
- type-name
- string-value
- typed-value
- is-id
- is-idrefs

Attribute Nodes must satisfy the following constraints.

- If an Attribute Node A is among the attributes of an element E, then the parent of A must be E.
- If a Attribute Node A has a parent element E, then A must be among the attributes of E.
- The data model permits Attribute Nodes without parents (to represent partial results during expression processing, for example). Such attributes must not appear among the attributes of any Element Node.

- In the node-name of an attribute node, if a namespace URI is present then a prefix must also be present.
- For convenience, the Element Node that owns this attribute is called its "parent" even though an Attribute Node is not a "child" of its parent element.

Accessors

dm:attributes

Returns the empty sequence.

dm:base-uri

If the attribute has a parent, returns the value of the dm:base-uri of its parent; otherwise it returns the empty sequence.

dm:children

Returns the empty sequence.

dm:document-uri

Returns the empty sequence.

dm:is-id

Returns the value of the is-id property.

dm:is-idrefs

Returns the value of the is-idrefs property.

dm:nilled

Returns the empty sequence.

dm:node-kind

Returns attribute.

dm:node-name

Returns the value of the node-name property.

dm:parent

Returns the value of the parent property.

dm:string-value

Returns the value of the string-value property.

dm:type-name

Returns the value of the type-name property.

dm:typed-value

Returns the value of the typed-value property.

dm:unparsed-entity-public-id

Returns the empty sequence.

dm:unparsed-entity-system-id

Returns the empty sequence.

B.1.4 Text Nodes

Text Nodes encapsulate XML character content. Text has the following properties:

- content
- parent, possibly empty.

Text Nodes must satisfy the following constraint:

- If the parent of a text node is not empty, the Text Node must not contain the zero-length string as its content.
- In addition, Document and Element Nodes impose the constraint that two consecutive Text Nodes can never occur as adjacent siblings. When a Document or Element Node is constructed, Text Nodes that would be adjacent must be combined into a single Text Node. If the resulting Text Node is empty, it must never be placed among the children of its parent, it is simply discarded.

Accessors

dm:attributes

Returns the empty sequence.

dm:base-uri

If the Text Node has a parent, returns the value of the dm:base-uri of its parent; otherwise, returns the empty sequence.

dm:children

Returns the empty sequence.

dm:document-uri

Returns the empty sequence.

dm:is-id

Returns the empty sequence.

dm:is-idrefs

Returns the empty sequence.

dm:nilled

Returns the empty sequence.

dm:node-kind

Returns text.

dm:node-name

Returns the empty sequence.

dm:parent

Returns the value of the parent property.

dm:string-value

Returns the value of the content property.

dm:type-name

Returns xs:untypedAtomic.

dm:typed-value

Returns the value of the content property as an xs:untypedAtomic.

dm:unparsed-entity-public-id

Returns the empty sequence.

dm:unparsed-entity-system-id

Returns the empty sequence.

C CellStore Performance Evaluation

A1	`/site/closed_auctions/closed_auction/annotation/description/text/keyword`
A2	`//closed_auction//keyword`
A3	`/site/closed_auctions/closed_auction//keyword`
C3	`/site/people/person[profile/@income = /site/open_auctions/ open_auction/current]/name`

Table C.1: Selected queries from the XPathMark benchmark

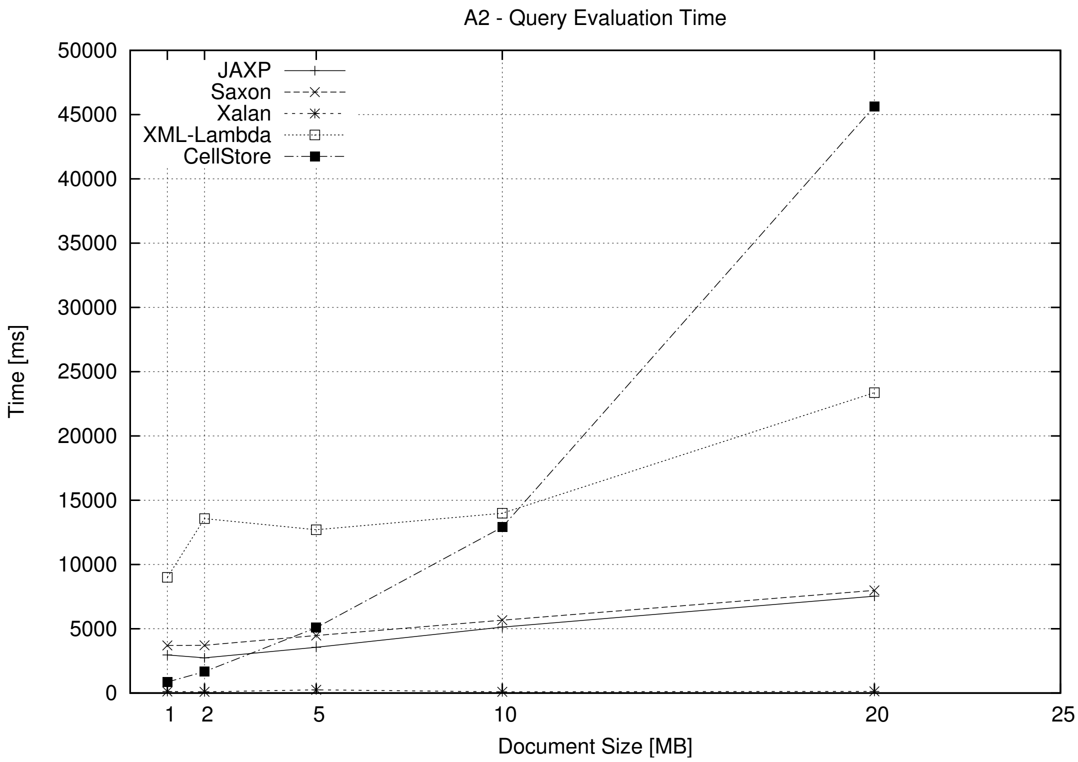


Figure C.1: *CellStore* A2 Query Performance

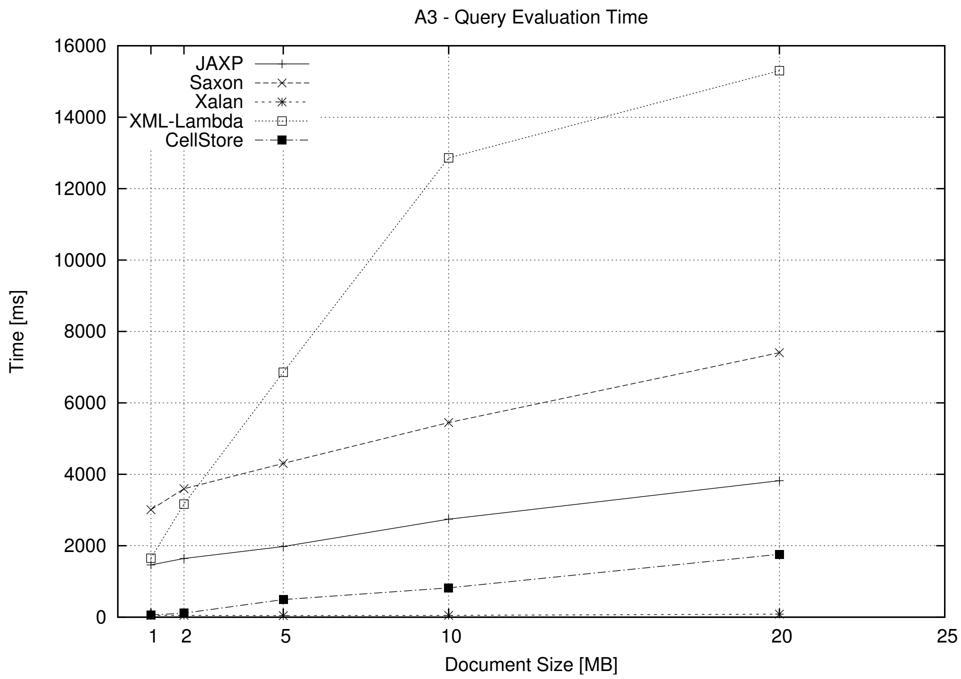


Figure C.2: *CellStore* A3 Query Performance

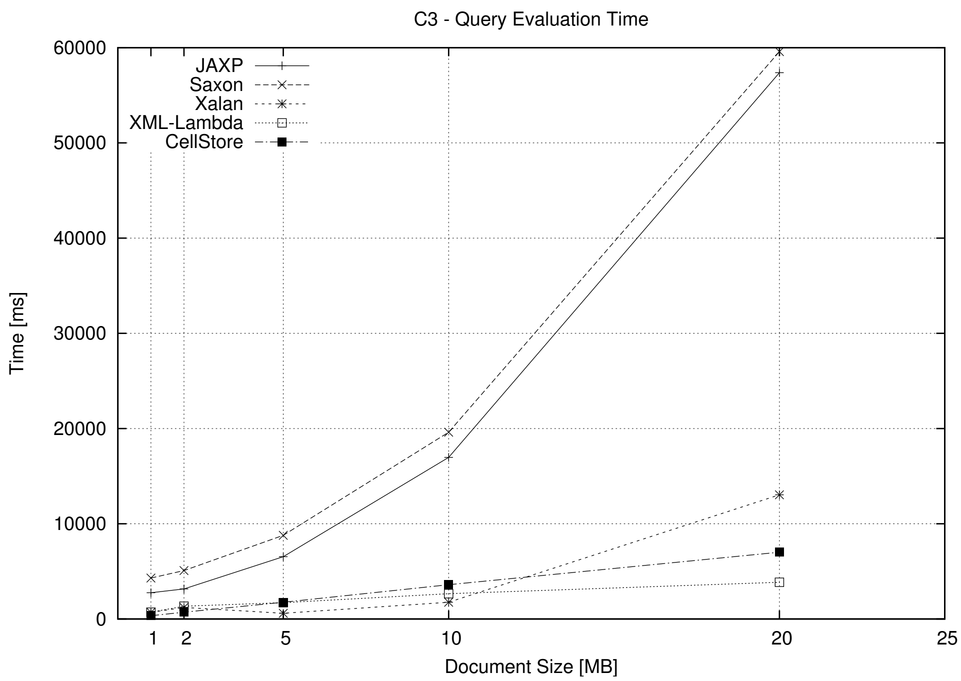


Figure C.3: *CellStore* C3 Query Performance

D Abbreviations

2PL Two-Phase Locking

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

CC Constraints Checker

CCL Constraints Check List

DOM Document Object Model

DTD Document Type Definition

EBNF Extended Backus-Naur Form

NXD Native XML Database

PUL Pending Update List

S2PL Strict Two-Phase Locking

W3C World Wide Web Consortium

XDGL XPath-based DataGuide Locking protocol

XDM XQuery 1.0 and XPath 2.0 Data Model

XLP XPath Locking Protocol

XML eXtensible Markup Languge

XPath XML Path Language

XQuery XML Query Language

XQUF XQuery Update Facility

XQUF-LP XQUF Locking Protocol

XSLT eXtensible Stylesheet Language Transformations

Index