Universität Konstanz

Chair of Databases and Information Systems
Department of Computer and Information Science

Bachelor Thesis

Fine Granular Locking in XML Databases

for obtaining the academic degree
Bachelor of Science (B.Sc.)

Field of study:	Information Engineering
Focus:	Systems Programming

by

Jens Erat
(01 / 692188)
jens.erat@uni-konstanz.de

First Examiner:	Prof. Dr. Marc H. Scholl
Second Examiner:	Prof. Dr. Marcel Waldvogel
Advisor:	Dr. Christian Grün

Konstanz, May 23, 2013

Acknowledgements

First of all, I want to thank Prof. Dr. Marc H. Scholl and the other members of the Database and Information Systems Group chair at the University of Konstanz for making this work possible.

I'm especially thankful to Prof. Dr. Marc H. Scholl and Prof. Dr. Marcel Waldvogel for being my referees and Dr. Christian Grün for advising and guiding me throughout both the accompagning project and this thesis.

Further I want to admit the large efforts accomplished by the whole BaseX team under lead of Dr. Christian Grün for making the software what it is now, it provided me an excellent foundation for the performance evaluations in my thesis. I want to highlight Leonard Wörteler, who not only has deep insight into the query core which he always was eager to share and also proofread my thesis.

Special recognition goes to my family for making my studies possible and always supporting me as needed.

Thank you all, und Dankeschön!

Abstract

XML databases gained highly in popularity over the last years, and queries performed got far more complex. Whereas mainly used for single threaded, often single user applications, usage for real-time, multi-user and parallel client-server environments increases. Along with that, demand for higher concurrency gets louder.

This bachelor thesis analyses requirements on and searches for suitable concurrency control algorithms suitable for the sequential XML encoding based on the pre/post plane widely used in native XML databases. For comparing different concepts, two of them have been implemented for BaseX - one of those native database systems:

- Conservative and strict two phase locking which was recognized as requirement to support all possible use cases, and
- optimistic concurrency control as a very different approach on achieving higher parallelism.

A short glimpse on other native XML database systems completes the evaluation of concurrency strategies.

While tree locking protocols have been dismissed, possible ways to further enhance concurrency control in BaseX are illustrated and considered.

Zusammenfassung

Über die letzten Jahre hinweg gewannen XML-Datenbanken stark an Bedeutung und die Anfragen auf immer größer werdende Dokumente wurden komplexer. Während früher vor allem in Single-Threaded- und Einzelbenutzerbetrieb genutzt wurden, verschiebt sich der Focus immer weiter zu Echtzeit-, Mehrbenutzer- und parallelen Client-Server-Systemen, mit welchen ein lauter werdender Ruf nach mehr Parallelität einhergeht.

Diese Bachelorarbeit erhebt Anforderungen an Sperrprotokolle und untersucht, welche Algorithmen für das verbreitete, auf der "pre/post-Plane" aufbauende sequenzielle XML-Encoding-Schema geeignet sind. Um verschiedene Konzepte zu vergleichen, wurden zwei von ihnen für BaseX, eines dieser Datenbanksysteme, implementiert:

- Das konservative, strikte Zwei-Phasen-Sperrprotokoll, welches als Voraussetzung erkennt wurde, um alle möglichen Anwendungsfälle abdecken zu können; sowie
- sogenanntes "Optimistic Concurrency Control" als einen deutlich anderen Ansatz um höhere Parallelität zu erreichen.

Ein kurzer Blick auf andere native XML-Datenbanksysteme schließt die Bewertung von Sperrprotokollen ab.

Während Baum-basierende Algorithmen als ungeeignet verworfen wurden, wird die Arbeit mit Überlegungen zu weiteren Verbesserungsmöglichkeiten abgeschlossen.

Contents

1 Introduction

1.1 Motivation

Since XML was specified in 1998, it was regarded as a format for small documents because of its bulky representation of stored contents. But data which has to be processed grew and progress in research allowed efficient retrieval from XML documents using query languages like XPath and XQuery. Soon, it was desired also to perform efficient updates, and XQuery Update was specified.

Native XML databases use special data structures offering efficient traversal of XML trees. There has been a lot of research on very fine granular locking on subtree and node level, but this is not required by all applications and does not need to be reasonable for all data structures and brings serious overhead on locking-effort.

1.2 Contribution

This thesis proposes ways to offer more fine granular locking based on collections of XML documents. After introducing XQuery Update and the relevant parts of data structures used by BaseX in the second chapter, I revisit common locking algorithms and consider them for usage inside BaseX. The fourth chapter briefly describes chosen implementations, which are put in contrast by benchmarking them against some elaborated use cases.

2 Preliminaries

2.1 XQuery

XQuery [Chamberlin et al. 2007] is a query language designed for retrieval and aggregation of collections of XML documents. It includes XPath expressions for accessing parts of the element tree, for example this statement printing the names of all German cities in the popular factbook document:[1]

```
//country[name="Germany"]//city/name
```

It is a functional language without side effects and mutable variables. Its high level constructs separate physical from logical representation.

So called FLWOR-expressions enhance it with SQL-like query capabilities for looping over result sets, extracting information from it, filtering and ordering results and constructing return values. They can be nested for performing joins and doing more complex calculations. Function declarations further enrich the language by allowing recursion and reuse of code and make XQuery a comprehensive, turing complete language.

An example for XQuery making use of a FLWOR-expression is this query calculating the number of people believing all religions sorted descending:

```
for $religion in //religions
let $name := $religion/text()
let $population := $religion/@percentage / 100
        * $religion/parent::country/@population
group by $name
order by sum($population) descending
return
  <population id="{␣$name␣}">
    { $population }
  </population>
```

2.1.1 Accessing Documents

XQuery knows several ways to access documents and document collections. Collections are sequences of arbitrary nodes.

[1] http://jmatchparser.sourceforge.net/factbook/.

- If no document is specified, the system dependent root context is used
- `fn:doc($uri as xs:string)` returns the data model representation of the document given by `$uri`
- `fn:collection()` returns the system dependent default collection
- `fn:collection($uri as xs:string)` returns the collection given by `$uri`

From here, we will consider a database as a collection or stored document not contained in a collection.

All these functions can be used as root of arbitrary subexpressions, thus multiple databases can be accessed in the same query, which enables joining them.

There are some other functions accessing the databases which eg. check if there is a document for a given URI. These match the signatures given above but for their return value and are not further elaborated here.

2.1.2 XQuery Update

While XQuery offers wide possibilities to access XML documents that were soon accepted as a popular way to query XML databases, it lacked a standardized possibility to permanently modify the underlying database.

In 2009, the W3C XQuery Update Facility [Chamberlin et al. 2009] was proposed as an extension to XQuery allowing to add, change and delete nodes in XML databases. For example, this query will insert the missing German city “Konstanz” into the factbook database:

```
insert node
  <city country="f0_220" province="f0_17529"
                             longitude="9.1" latitude="47.4">
    <name>Konstanz</name>
    <population year="11">85524</population>
  </city>
into //country[name="Germany"]/province[@name="Baden␣Wurttemberg"]
```

As XQuery is a declarative language prohibiting side-effects, changes performed with XQuery Update are not directly applied to the database, but first collected in the “pending updates list”. After successfully evaluating a query, the database is modified. I will later exploit this aspect of XQuery Update.

2.2 XML Encoding Schemes

For being able to do efficient query processing on XML documents, a suitable representation must be found. Well-known APIs for accessing XML like SAX and DOM are inappropriate for obvious reasons like either enforcing serial evaluation or the need to fit all the document into the main memory which puts a severe limit on the processable document size.

In general two different approaches exist:

Hierarchical numbering, which extends the parent's node ID by sub-IDs for the children. Compounded, they form a variable-length unique identifier for each node like `1.3.1`. ORDPATH [O'Neil et al. 2004] solves the problem of having to renumber all subtrees of following nodes when inserting or removing elements through exclusively using odd numbers for children, while even numbers stand for inserted nodes on the same level, so inserting between `1.1` and `1.3` leads to the node ID `1.2.1`, as visualized in figure 2.1.

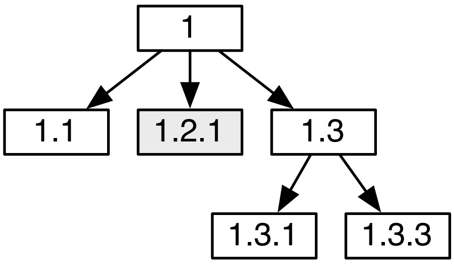
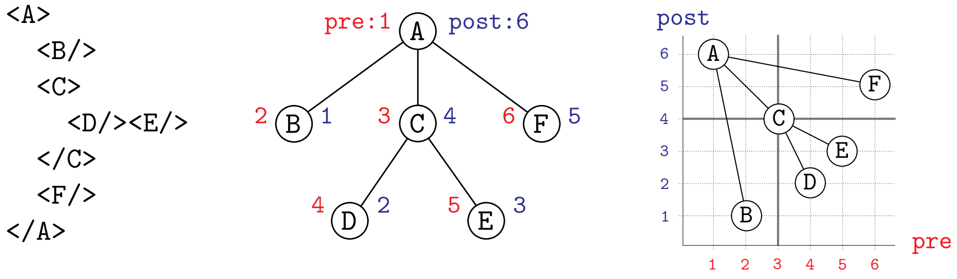


Figure 2.1: ORDPATH insertion example

Storing and comparing these variable-length node IDs is difficult and expensive, thus there was search for another representation with fixed-length IDs.

Sequential numbering exploits that trees can be reconstructed by combining different traversal strategies like pre and post-traversal, which results in fixed length keys. [Grust 2002] showed that when using pre and post IDs, each node partitions the pre/post-plane into four disjoint regions, see figure 2.2. An XPath step on the four major axes ancestor, descendant, preceeding and following nodes is equivalent to retrieving all elements in one of these regions, which is a query that can be efficiently implemented using additional tree structures.



```
<A>
  <B/>
  <C>
    <D/><E/>
  </C>
  <F/>
</A>
```

Figure 2.2: XML document with pre/post values and in pre/post plane [Grün 2010:9]

A drawback of this encoding scheme is that inserting or removing a single node can require changing IDs for big parts of and even the whole tree. A slightly changed representation which will be discussed in the next section reduces the number of nodes to be changed significantly.

2.3 BaseX

The native XML database BaseX was initially developed by Christian Grün as practical offspring while doing research for his thesis on *"Storing and Querying Large XML Instances"* [Grün 2010]. Since then, it was further enhanced by the Database and Information Systems group at the University of Konstanz, led by Professor Dr. Marc H. Scholl, and the newly-founded BaseX GmbH.

It offers a highly performant implementation of the W3C XQuery 3.0 Recommendation including its Full Text and Update extensions. With great graphical user interfaces, APIs for multiple programming languages and a RestXQ-implementation it is wide-spread used in stand-alone, embedded, client-server and web applications in both industry and research.

Up to now, BaseX does not support transactions spanning multiple queries. In the following parts of my thesis, the terms "query" and "transaction" should be considered equal unless emphasized specially.

As part of the project accompanying the thesis, I enhanced BaseX' locking capabilities by implementing strict and conservative two phase locking on database level. Before, all writing transactions required a system-wide exclusive lock blocking all other concurrent operations.

2.3.1 Core Architecture

Each running BaseX instance has one context storing information on the database system's state. Each command to be executed holds references to this context. Commands can be administrative commands like `COPY` used in the architecture overview diagram (figure 2.3).

XQuery queries are also implemented as a command, `XQUERY`. It has a root expression attached, which references one or more subexpression possibly having subexpressions themselves, forming an abstract syntax tree.

Before executing, a command `register()`s itself with the context and `unregister()`s after committing.

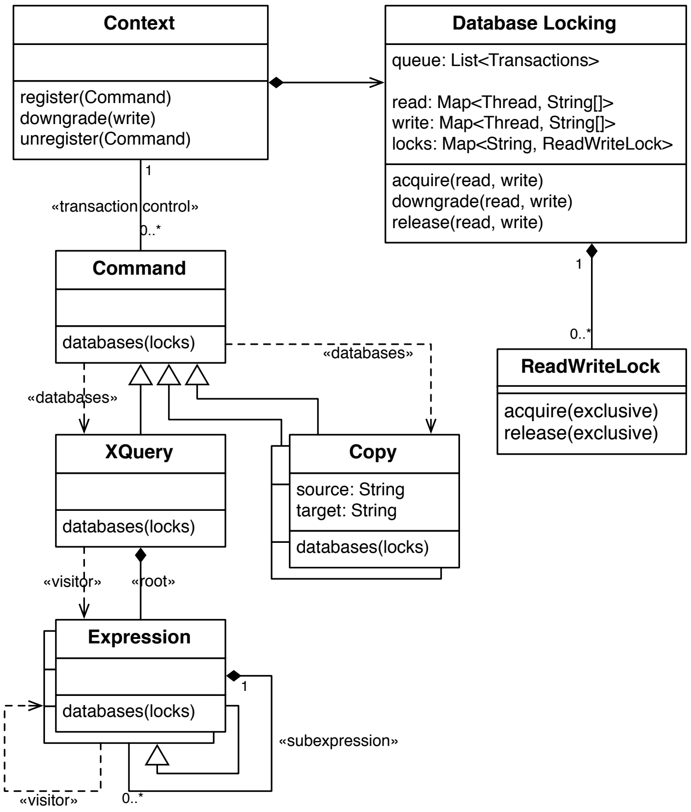


Figure 2.3: Core Architecture

2.3.2 Pre/Dist/Size Mapping

Grün proposed the pre/dist/size sequential mapping as an evolution of the pre/post mapping reducing the number of nodes that have to be modified when inserting or deleting other nodes. It has some advantages over the pre/post approach, which are discussed in detail in [Grün 2010:14].

Pre-Value Node-ID in pre-order, sorted by document order and implicitly stored by row number

Dist-Value Difference between the node's pre-value and its parent's pre-value

Size-Value Number of nodes in this node's subtree

Using these values, all XPath axes can be traversed efficiently using algorithms inspired by the staircase join [Grün 2010:83]. As details on this are not important to my work, I will refer to the thesis for more details.

Most important for updates is the use of relative references, so that only some nodes have to be updated when changing the document structure [Kircher 2010]. The pre-value changes implicitly by moving the records. All dist-values on the changed node's following-sibling-axis must be updated, also the size-values on all nodes on the ancestor-axis to represent the new subtree's size.

While this usually limits the number of nodes to changes to very few, updating the ancestor-axis *always includes the root node*, which will be important for discussing locking algorithms.

3 Concurrency Control Algorithms

In this chapter, I will review requirements to concurrency control and then compare different algorithms for applicability in native XML databases.

I distinguish between two kinds of concurrency control protocols, document and tree based ones. Document based concurrency control regards each tree or a forest of trees as a separate object, tree based algorithms look into these and lock on subtrees or nodes.

3.1 Requirements

Every transaction should adhere the principal "ACID" properties of transactions (Atomicity, Consistency, Isolation and Durability) [Haerder and Reuter 1983] to prevent update anomalies. I will discuss those prinipals in following sections.

Due the power and flexibility of XQuery, transactions run in those system often get rather complex up to whole applications executed as database queries including side effects beyond scope and control of the database systems, for example

- HTTP POST and REST requests changing external state,
- writing to the file system and
- access to other database management systems.

These external effects add further limits on choosing appropriate algorithms.

3.1.1 Atomicity

This aspect requires transactions to either finish completely and successfully or have no effect to the database at all. If an error occurs, the whole transaction must be aborted and changes rolled back if necessary.

Read only transactions do not modify the underlying storage and thus need not be considered further.

XQuery Update's "pending update list" mentioned above further helps ensuring atomicity of single query transactions, as changes are only written to the storage if the query was evaluated successfully. When errors occur while evaluating the query, no rollback is needed as the data store was not modified yet. Yet transactions spanning multiple queries require some kind of rollback mechanism.

Other encounters to atomicity are crashes beyond query execution scope because of software and hardware failures or power outages.

Usual methods for dealing with problems harming atomicity are logging modifications or storing multiple versions. BaseX does not yet support rollbacks and thus does not ensure atomicity in case of backend failures.

3.1.2 Consistency

This aspect requires the database to be in a consistent state after the execution of each transaction. For XML databases, this includes

- no errors in the underlying representation,
- each document being well-formed XML and
- if DTDs or XML Schemes are supported, that documents adhere to those.

For ensuring consistency, BaseX validates the pending update list on producing valid XML documents before applying it. DTDs and XML Schema are not supported yet, but could be validated the same way.

3.1.3 Isolation

This is the most important aspect regarding concurrency control. Isolation must prevent transactions executed in parallel from having impact on each other. The system state obtained after executing transactions in parallel must be the same as when executing them in serial order.

Before my project accompanying this thesis, BaseX ensured isolation by using a process lock preventing a updating transaction from running in parallel with any other transaction. This was enhanced to locking on database level which will be presented in another chapter. Now updating transactions only block the databases they're accessing.

3.1.4 Durability

Durability ensures that changes made by a successfully committed transaction is persistent also if system failures (hardware or software) occur. This especially includes malfunctions in the volatile main memory and power outages.

In BaseX, durability is ensured as long as all writes are automatically flushed to secondary storage this can be disabled for performance reasons. It does not support incremental, online backups to tertiary storage.

3.1.5 External Side Effects

As mentioned in the introduction to this section, XQuery is also used for complex applications also accessing other resources. If those have side effects that cannot be rolled back by the database system – which applies to most of the external side effects –, transactions may not be restarted.

Supporting this kind of complex queries is especially important as XQuery and native XML databases are gaining a growing importance as application servers for business logic and web services, which are easy to implement using APIs like RESTXQ [Retter 2012].

3.2 Process Locking

The database management system supports multiple reading transactions at the same time, but no writing transactions, regardless whether they access common resources or not. This was implemented by BaseX up to now [Weiler 2010].

3.3 Document Based Locking Protocols

These protocols are applied on trees or forests as locking objects.

In some XML database systems including BaseX, a forest of multiple XML trees inside the same collection are stored in the same data structure.

3.3.1 Two Phase Locking

Two phase locking is a very common concurrency protocol. It distinguishes between two phases,

1. the **expansion phase** in which locks are acquired and
2. the **shrinking phase** in which locks are released.

Splitting up locking in these two phases ensures serializability.

In XQuery Update, all changes are performed after fully evaluating the query. This implicitly enforces strict two phase locking, where no write locks may be released before the end of the transaction.

Two phase locking, also in its strict version, can result in deadlocks when two transactions crosswise try to acquire a lock already held by the other one. If a deadlock occurs, one of these transactions must be aborted and possibly restarted. As stated earlier in the section about side effects, this might not be acceptable for all transactions. Deadlocks can be prevented by using **conservative two phase locking** [Weikum and Vossen 2001:142] which preclaims all locks at the beginning of the transaction. This can bring a large penalty on concurrency if it is impossible to determine which databases to lock, resulting in global locks enforcing serial execution.

3.3.2 Multi Version Concurrency Control

Especially in its conservative and strict variants holding locks for long parts of the transaction, two phase locking suffers from low parallelism.

A common solution to this problem, especially used in the relational database world, is multi version concurrency control. When performing updates, the old version of the data is not overwritten, but a copy is created and modified. A read-only transaction always performs on the newest committed version. Old versions are subject to garbage collection as soon as the last transaction reading them are terminated. Read-only transaction are not subject to blocking because of waiting for writing transactions working on the same object any more, and writing transactions only need to wait for other writing transactions on the same object.

3.3.3 Optimistic Concurrency Control

Contrary to the algorithms presented before, optimistic concurrency control [Kung and Robinson 1981] does not rely on locks to ensure serializability. Opposed to these “pessimistic” protocols, optimistic concurrency control expects conflicts to occur rarely.

Optimistic concurrency control distinguishes between multiple phases, namely

1. **Read phase**: Execute transaction, but keep all writes in a private workspace. At the end of the transaction, both write and read sets are known.
2. **Validation phase**: When a transaction is ready to commit, it is checked for serializability. For this purpose all accessed objects (both read and write) are tested on being changed after the transaction was started. A change means that the transaction might have read inconsistent data and must be aborted (and possibly restarted) due to violating serial execution. Reading transactions are committed at the end of this phase.
3. **Write phase**: If a transaction was successfully validated, its write set gets applied. After writing, a transaction is committed.

Validation and write phase should be executed in an exclusive fashion. For testing on changes, each transactions gets assigned an incremental, unique time stamp. Each object stores the timestamp of the last transaction writing it, this gets updated during the write phase. If an object timestamp is larger than the transaction timestamp, it got modified after the transaction was started.

There are two kinds of optimistic concurrency control.

- **BOCC** (backward-oriented concurrency control) validates after the read phase is finished. If validation fails, the transaction being validated gets restarted.

- **FOCC** (forward-oriented concurrency control) validates all other running transactions that are in the read-phase. If any other transaction read a database that the validating transaction wants to write, the other transaction is restarted.

Usually, FOCC is to be preferred as it is fail-fast (it aborts transactions that will fail to validate before they complete the read phase).

Interrupting all other transactions during the validation phase of a committing transaction can be difficult. When using FOCC it is sufficient to prevent transactions in the read phase from gaining access to objects they didn't access before, which can easily be implemented using a semaphore. Transactions already having access to objects that shall be written will be aborted anyway.

Regarding the write phase exclusive execution is only important for objects which the transaction will write to. This can easily be implemented using conservative two phase locking with locks acquired during validation phase.

While keeping changes in a private workspace and only applying it immediatly before committing a transaction seems as a bothersome overhead, this does not apply to XQuery Update as the pending update list does exactly do this anyway.

3.4 Tree Locking Protocols

It is common in relational database management systems to lock on record and page level to prevent locking whole tables or databases. However, nodes in an XML documents which are all connected cannot be directly compared to records in RDBMS systems which mostly aren't related to each other.

Queries on *relational databases* usually affect single records, ranges of some attribute or complete tables for aggregation. Queries on *XML databases* usually have effect on whole subtrees and require a path between the root nodes of the document and that subtree, so the accessed node sets usually overlap.

However, fine granular locking is highly desired for achieving better concurrency, and lots of research is going on in this area.

Helmer et al. [2001] proposed different algorithms locking on varying granularity and evaluated them in [Helmer et al. 2004], namely Doc2PL, Node2PL, NO2PL and OO2PL.

3.4.1 Doc2PL

While document two phase locking actually equates to the two phase locking discussed above locking collections, it can also be regarded as the most simple tree locking protocol.

3.4.2 Node2PL

Node two phase locking puts a read lock on every node accessed when reading or traversing it. Exclusive locks on a node to be modified are put on the parent node. This results in locking complete subtrees.

3.4.3 NO2PL

In contrast to Node2PL, NO2PL does not lock whole subtrees, but only nodes containing pointers that get modified, especially the parent, sibling and children nodes. This permits higher concurrency than locking the whole subtree.

3.4.4 OO2PL

Whereas the former two protocols lock on nodes, OO2PL locks pointers between them. It distinguishes between four kinds of pointers between nodes, first child (A), last child (Z), left sibling (L) and right sibling (R). There are shared traverse locks (T) and exclusive modify locks (M) for each of these.

Locks are acquired for each node traversed or modified and can be shared based on a compatibility matrix.

3.4.5 taDOM-Tree

The taDOM tree proposed by Hausetin Haustein and Härder [2003] extends the DOM tree by two elements. It moves attributes below an attribute root node which exists for all elements and the new “string” node.

Different concurrency control mechanisms on this tree have been proposed in Haustein and Härder [2003] and Haustein and Härder [2004].

3.4.5.1 Node Locks

Node locks are comparable to NO2PL. Haustein proposes a set of seven partially compatible lock types on nodes allowing different kinds of operations. When traversing nodes, these must be locked with matching locks for reading or updating operations.

Because of the newly introduced subnodes attribute root and strings, higher concurrency for updates on attributes and values is possible compared to NO2PL.

3.4.5.2 Navigation Locks

Navigation locks are comparable to OO2PL. Again, locks are put on edges. taDOM navigation locks on the same navigation paths as OO2PL does, but only distinguishes between three different lock types.

Like in taDOM node locks, the new subnodes move some locks deeper down in the tree and thus offer higher concurrency.

3.5 Comparison and Applicability

Tree locking protocols trade more overhead on locking for higher concurrency as they're locking on a more fine granular level than document based protocols, but require many more individual locks. Difference in throughput achieved is negligible between the node (NO2PL, taDOM node locks) and edge (OO2PL, taDOM navigation locks) based protocols [Haustein and Härder 2004].

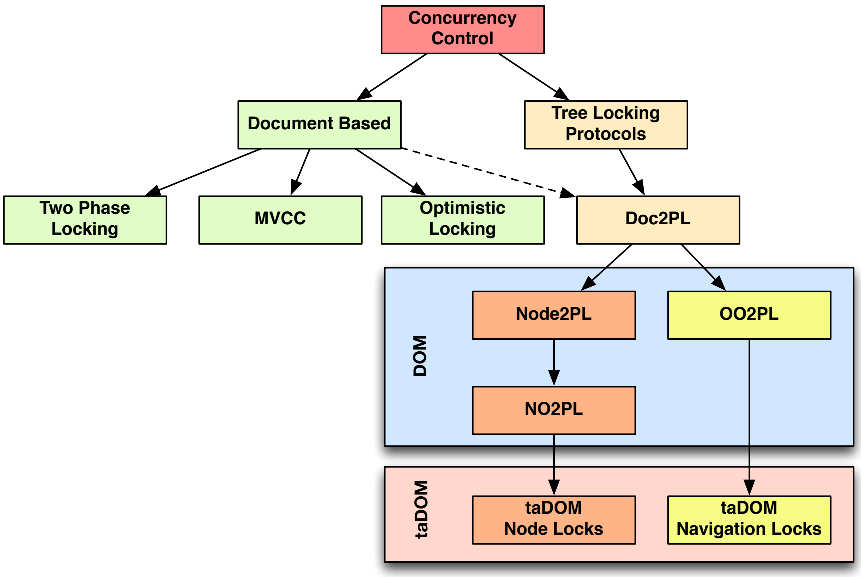


Figure 3.1: Comparison of Concurrency Control Algorithms

As discussed in the XML encoding section, there are mainly two different approaches on representing XML documents for XML databases. Haustein [2005], p. 95 discovered that only the hierarchical numbering schemes ORDPATH and the related DLN support all requirements on tree based fine granular locking, however no sequential schema does.

It is important to recognize that applying tree locking protocols with sequential numbering schemes is not reasonable anyway, as at least the ancestor-axis must be exclusively locked including the root node. As a consequence, all tree based protocols degrade to document based locking, but their tremendous effort on fine granular locking stays.

Therefore, only document based concurrency protocols are reasonable candidates in sequential numbering based systems like BaseX.

When recalling the requirements raised above, it becomes clear that only conservative two phase locking, possibly applied within multi version concurrency control, can avoid the need to abort transactions because of deadlocks; so one of these must be chosen as basic concurrency control protocol. Yet it is desirable to allow more relaxed locking as it can highly increase concurrency, ideally at the same time with conservative locking for transactions with uncontrolled side effects. This could be implemented by either automatically recognizing side-effect free transactions or leaving it to the developer to decide on how the transaction is executed, for example by declaring an XQuery option.

4 Implementation of Concurrency Control Protocols in BaseX

As conservative two phase locking was determined as the only option without totally enforcing serial execution in systems with some queries having external side effects, conservative strict two phase locking was chosen. While most queries would benefit from giving up conservativeness, this is not implemented yet due to time constraints and kept as future work.

For comparing to a radically different approach on achieving higher parallelism, I implemented a preview version optimistic concurrency control with limited capabilities, especially support for administrative commands is missing.

4.1 Conservative Strict Two Phase Locking

Central component is the DBLocking class which manages read/write lock instances for all databases and the acquisition and release of locks requested by transactions. It is attached to the database context, where it hooks into the `register()` and `unregister()` calls.

Each command implements a function to declare which databases to lock, distinguishing between read-only and write access. `COPY` for example will declare reading from `source` and writing to `target`.

The abstract syntax tree of XQuery expressions is investigated using a visitor descending all paths. All subexpressions accessing databases add those. If one cannot determine which to lock at compile time, investigation is stopped and all databases are locked instead. The visitor is started by the `XQUERY` command wrapping the query. Inlining and other optimizations to reduce the number of locked databases can be performed prior to visiting the syntax tree.

4.1.1 Database Locking Class

This class manages a queue of transactions to limit the number of concurrent transactions sharing the system's resources to a reasonable number. When locks are acquired, they're stored in maps with the thread (transaction) as key to retrieve them while releasing later. For each database, one lock is managed.

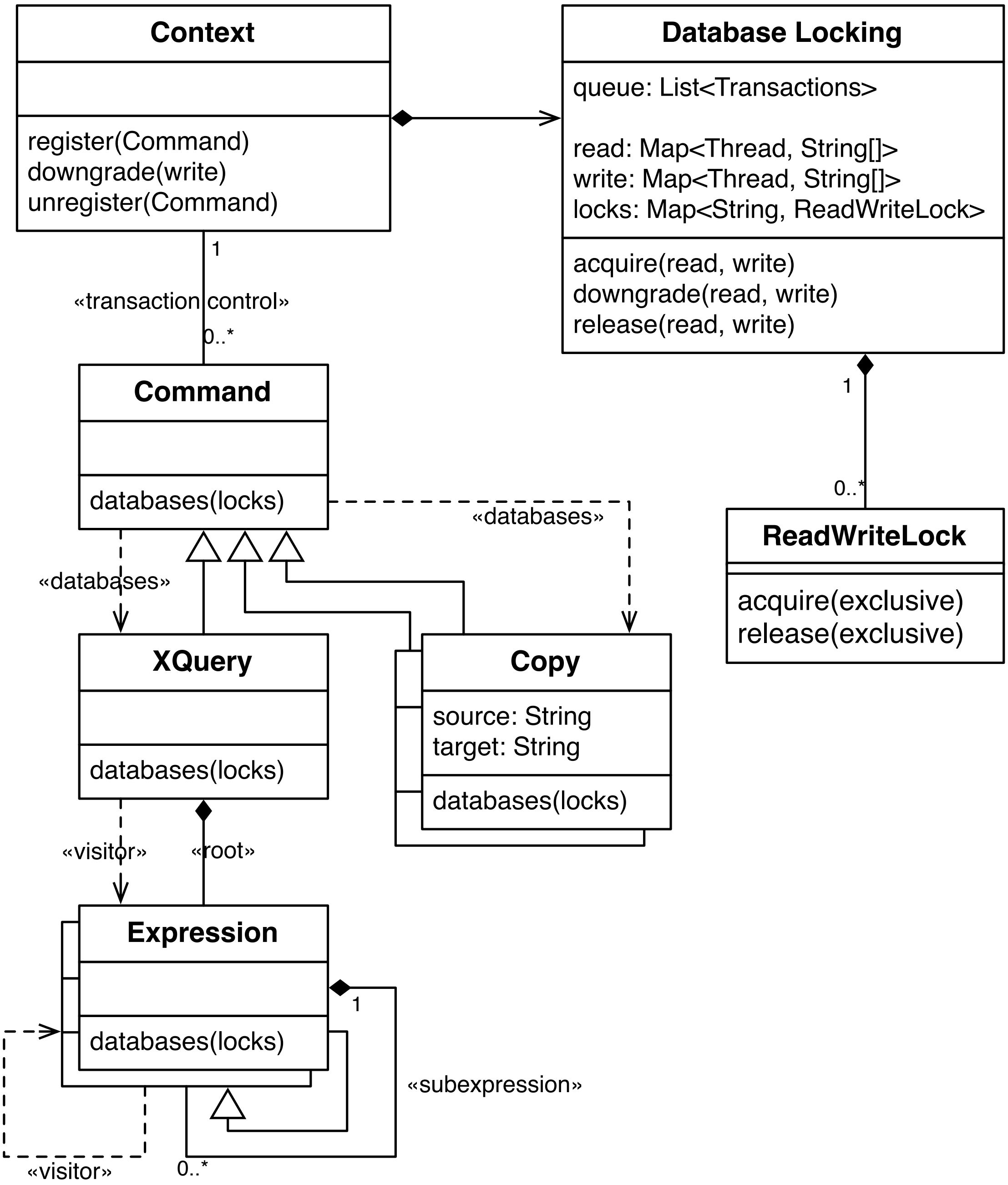


Figure 4.1: Class diagram for 2PL implementation

4.1.1.1 Acquire

Locks are acquired in the beginning of the transaction.

```
function ACQUIRE(read locks, write locks)
    wait-in-queue()                                   ▷ Limit amount of concurrency

    if unknown(write locks) then                      ▷ Acquire global locks as needed
        acquire(global write lock)
    end if
    if unknown(read locks) then
        acquire(global read lock)
    end if
                                        ▷ Ensure order on locks to prevent deadlocks
    write locks ← unique(sort(write locks))
    store(write locks)
    read locks ← unique(sort(read locks))
    store(read locks)

    locks ← sort( {write locks, read locks} )                 ▷ Acquire local locks
    for all locks as lock do
        acquire(lock)
    end for
end function
```

4.1.1.2 Downgrade

Before applying the pending update list, downgrade will be called with the now complete list of databases to write to.

```
function DOWNGRADE(new write locks)
    new write locks ← unique(sort(new write locks))
                                             ▷ Retreive locks previously acquired
    downgraded write locks ← fetch(write locks) \ new write locks
    write locks ← new write locks
    read locks ← fetch(read locks) ∪ downgraded write locks
    for all downgraded write locks as lock do              ▷ Downgrade local locks
        atomic:
          release(lock.write)
          acquire(lock.read)
    end for

                                                  ▷ Downgrade global write lock
    if unknown(downgraded write locks) ∧ known(new write locks) then
        atomic:
            release(global write lock)
            acquire(global read lock)
        end if

        store(write locks)
        store(read locks)
    end function
```

4.1.1.3 Release

After committing the transaction, all locks are finally released. The locks are retrieved from the maps they have been stored to earlier.

```
    function RELEASE
        fetch(write locks)                          ▷ Retreive locks previously acquired
        fetch(read locks)

        for all {write locks, read locks} as lock do           ▷ Release local locks
            release(lock)
        end for

        if unknown(write locks) then                          ▷ Release global locks
            release(global write lock)
        end if
        if unknown(read locks) then
            release(global read lock)
        end if

        if not empty(queue) then
            notify(next transaction in queue)
        end if
    end function
```

4.1.2 Determining Databases to Lock

Conservative two phase locking requires knowledge of accessed databases in advance. While determining touched databases for non-XQuery operations like administrative commands is trivial, this is much more complicated for complex XQuery statements. Apart from the standard XQuery functions to access databases, most systems offer further administrative functions that need to be regarded on their own.

Databases which will be accessed are passed as a string parameter to all these functions. This can be an explicit string like `doc('factbook')` which is rather easy to determine, but can also consist of arbitrary subexpressions containing function calls and variables, for example `doc('user' || $i)`. Even more complexity can be added by declaring variables or looping over sequences of documents to access:

```
for $db in ('foo', 'bar')
return doc($db)
```

Often, such constructs can be precomputed or inlined as static optimizations during query compilation time.

However, there are queries that prevent calculating the accessed database set before runtime. The database name could also be passed as external variable (`declare variable $db external; doc($db)`) or even depend on the content of other databases:

```
let $db := doc('foo')/bar)
return doc($db)
```

If it is not possible to determine all accessed databases in advance, conservative two phase locking is not possible without locking all available databases.

4.2 Optimistic Concurrency Control

Optimistic concurrency control was implemented in it's forward-optimistic flavor with an optimized validation and write phase.

When a transactions reaches the validation phase and is updating, all other running transactions are checked whether they already accessed the databases to be modified. Opposed to proposed in literature, the validation phase only blocks other transactions when opening new databases, but keeps them running as long they're not interfering with the validating transaction.

The exclusiveness condition during the write phase was relaxed, too. Parallel transactions are allowed, but access is controlled by conservative two phase locking.

There are two validation paths, which are executed mutually exclusive.

Restarted transactions wait until the transaction aborting them committed.

4.2.1 Validation on First Access

When a subexpression accesses a database for the first time, it already compares the transaction IDs for the first time. This is necessary to prevent reading access after another transaction started its validation and write phase. This validation occurs during *read phase.*

```
function VALIDATE-DATABASE(transaction, database)
    if transaction.id < database.last-id then
        throw QueryRestartException
    end if
end function
```

4.2.2 Validation Phase

During validation phase, a transaction verifies no other transaction read or is still reading a database to be written by the transaction currently running. This validation occurs during *validation phase*.

```
function VALIDATE(transaction, databases)
    for all databases as database do
        database.last-id = transaction.id
    end for

    for all transactions as other-transaction do
        if other-transaction.id < database.last-id then
            throw QueryRestartException in other-transaction
        end if
    end for
end function
```

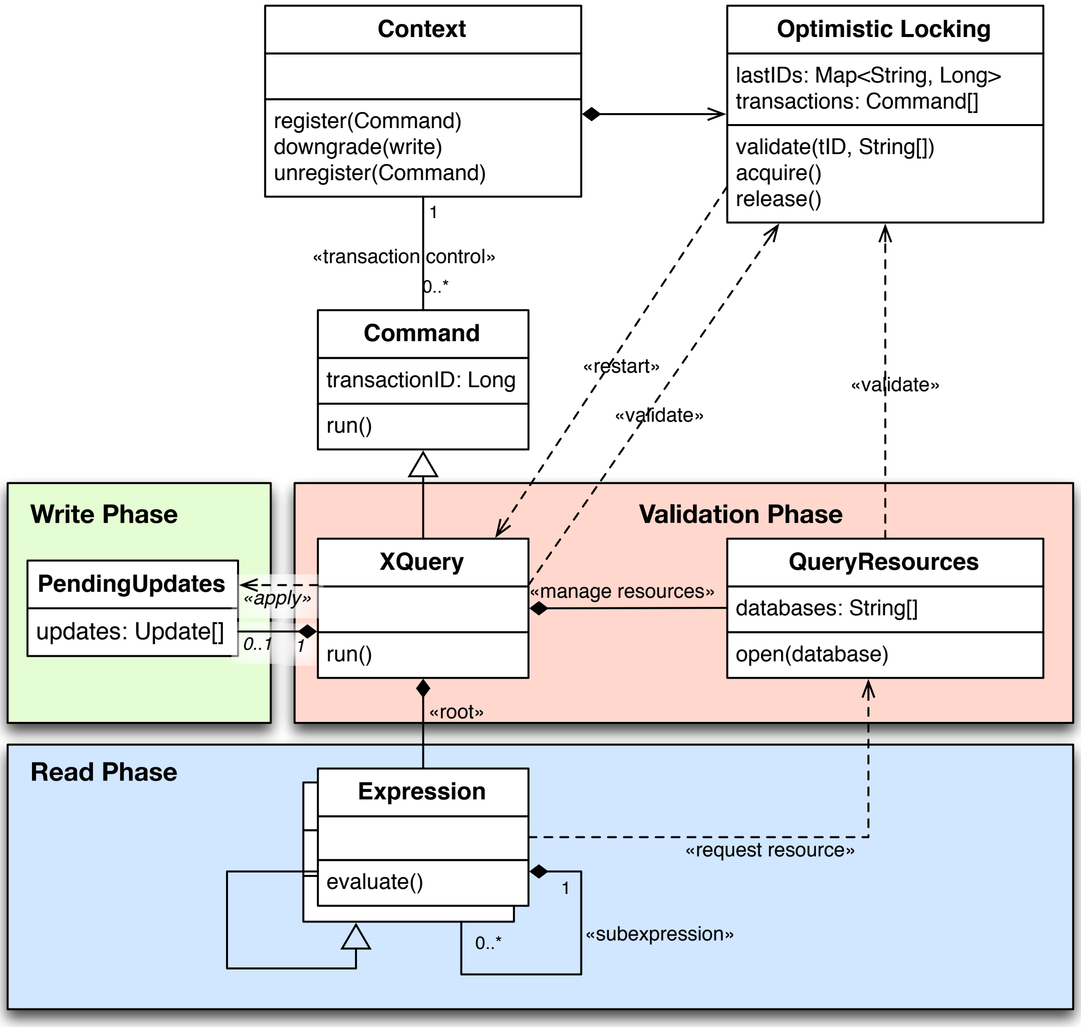


Figure 4.2: Class diagram for optimistic concurrency control implementation

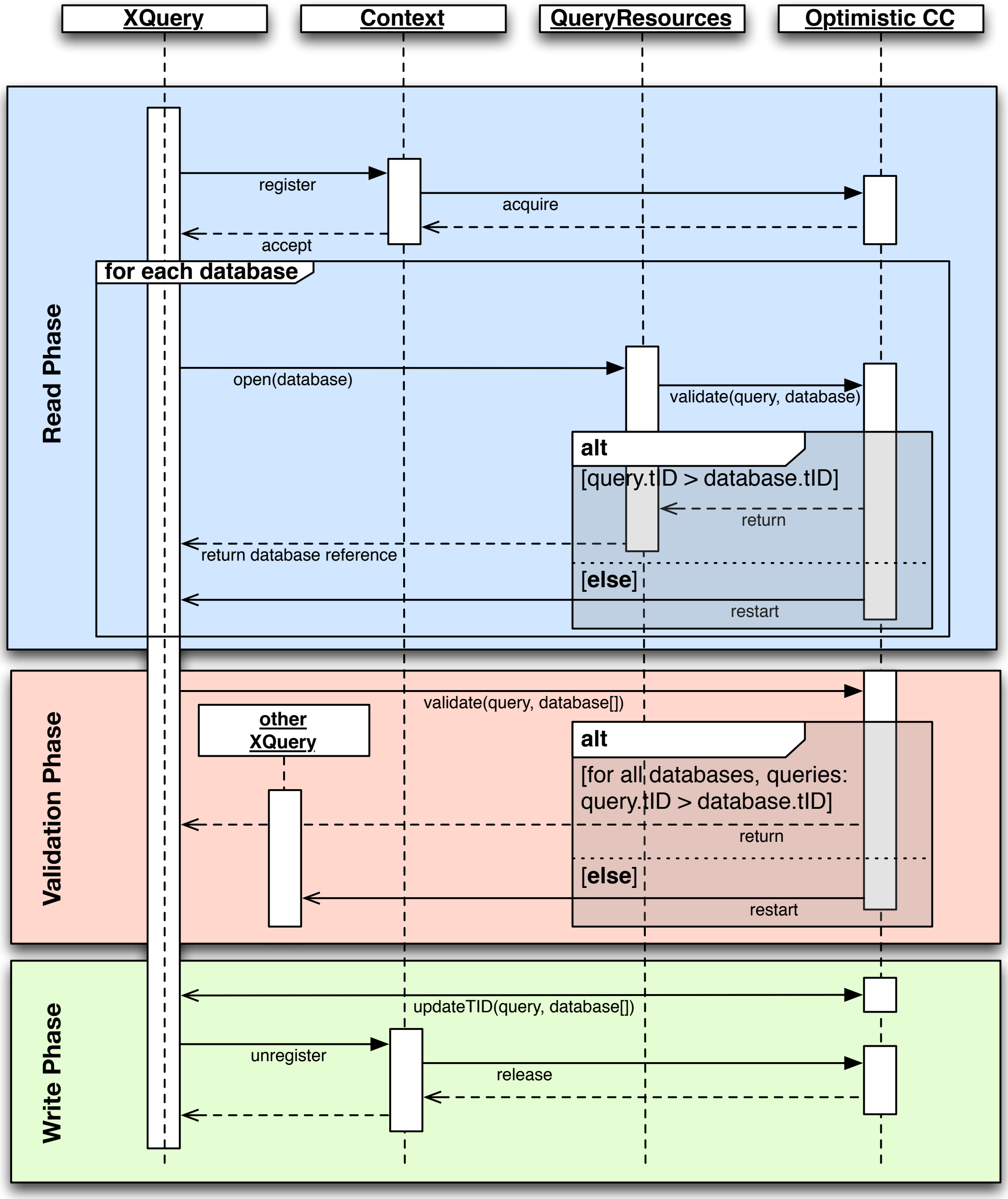


Figure 4.3: Sequence diagram for optimistic concurrency control implementation

5 Benchmarks

There is a vast amount of locking algorithms, all with different advantages and drawbacks. Some of them might be critical in some applications while irrelevant in others, and the range of basic conditions and general requirements is huge. For this reason, it is important to evaluate these algorithms against typical use cases.

For each use case, one should consider

- Typical query sets specified by
  - whether they're updating (requiring exclusive execution),
  - how often it is run compared to others,
  - the execution time and
  - which resources are accessed
- External side effects which prevent rolling back a query

For these benchmarks external side effects are neglected; they would only limit choice on available algorithms without changing results for the suitable ones.

For benchmarking, *process locking*, *conservative two phase locking* and *optimistic locking* are put in contrast. As the aim is to benchmark the performance of the concurrency control algorithms and not the hardware or query processor, the used queries are reduced to an extend which requires minimum access to databases:

- **Reading queries** return the number of elements in a database, which requires opening it, but is evaluated based on index statistics in constant time [Grün 2010:119].
- **Writing queries** append an element to the end of the database, limiting accessed nodes to a minimum.

To simulate different complexities in run time, BaseX' propietary XQuery function `prof:sleep($ms as xs:integer)`[1] is used.

If the database name should be known in advance, it was referenced directly using `doc('db-name')`, otherwise by running `let $db := 'db-name' return doc($db)` which is not yet optimized for determining the database name.

Example queries used for benchmarking are:

[1] Http://docs.basex.org/wiki/Profiling_Module#prof:sleep.

- `(count(doc('db')//*), prof:sleep(100))`
- `insert node (<a>{count(doc('db')//*)}</a>, prof:sleep(100)) into doc('db')`

The queries are run in a semi-shuffled order determined by a constant seed warranting reproducible execution. For running the queries, eight parallel client threads sent the queries to the database system. This number was chosen as the default limit on parallel execution in BaseX as determined as a reasonable limit for most systems and request profiles. Validating on different numbers was considered but abolished as it would mainly benchmark the system's power.

As there can be a huge difference in waiting time especially between transactions directly passing locking – often forming the largest portion –, a logarithmic scale was chosen in the diagrams.

5.1 Corner Cases

5.1.1 CC1: Parallel Reading

This corner case performs lots of small reading transactions. These can all be executed in parallel and should reveal any remarkable locking overhead.

5.1.1.1 Queries

1. Read from database
   - Updating: No
   - Frequency: High
   - Runtime: Short
   - Locking: Same single database

5.1.1.2 Benchmark

As might be expected, no huge difference is to be noted in Figure 5.1. The number of outliers equals in all implementation with only optimistic concurrency control having a small advantage. This can be explained by a smaller chance to be hit by garbage collection and other side effects due to less CPU time spent in lock acquisition.

While one could expect a minor penalty for the more complex locking code in database locking, this seems negligible and has an effect in fractions of milliseconds.

5.1.2 CC2: Parallel Writing, Same Database

This corner case performs lots of small writing transactions to different databases. They need to be executed serialized and should reveal any markable locking overhead.

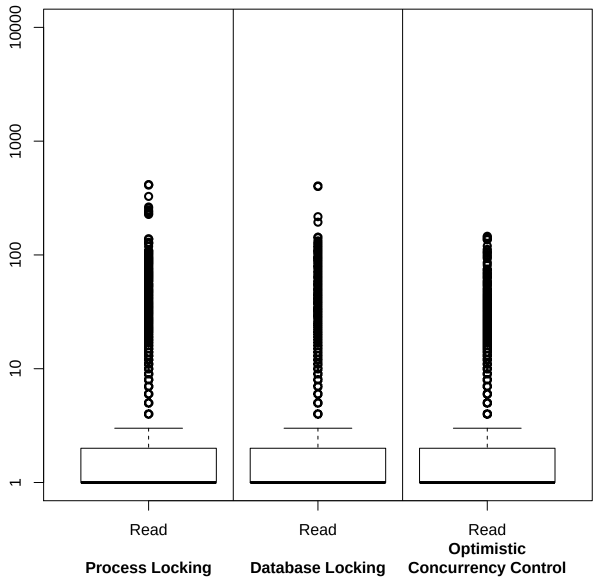


Figure 5.1: Waiting Time for CC1: Parallel Reading

5.1.2.1 Queries

1. Write to database
   - Updating: Yes
   - Frequency: High
   - Runtime: Short
   - Locking: Same single database

5.1.2.2 Benchmark

Figure 5.2 does not show any surprises, either. The queries are executed in serial order, so waiting time equals about 700ms - which is the execution time multiplied by the number of parallel queries allowed minus one. This is the number of transactions waiting on the lock or to be restarted in case of optimistic concurrency control.

5.1.3 CC3: Parallel Writing, Different Databases

This corner case performs medium sized writing transactions to different databases. They can be executed in parallel. Databases to write can be calculated at compile time.

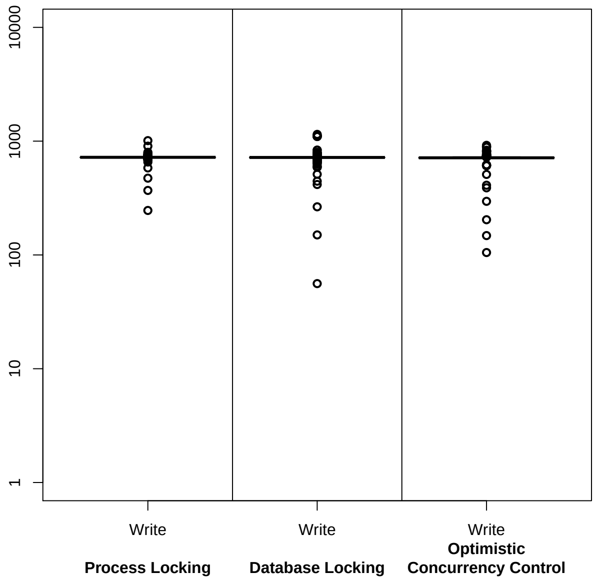


Figure 5.2: Waiting Time for CC2: Parallel Writing, Same Database

5.1.3.1 Queries

1. Write to database
   - Updating: Yes
   - Frequency: High
   - Runtime: Medium
   - Locking: Varying single database, known in advance

5.1.3.2 Benchmark

Figure 5.3 clearly shows the advantage of more sophisticated concurrency control algorithms. As the queries are mostly writing to different databases, they rarely need to wait - in contrast to process locking whose box plot resembles the one of CC2. Calculation time wasted before being restarted is included in waiting time, which explains the slightly larger spread for optimistic concurrency control.

5.1.4 CC4: Parallel Writing, Different Databases Unknown at Compile Time

This corner case performs medium sized writing transactions to different databases. They can be executed in parallel. Databases to write are unknown at compile time.

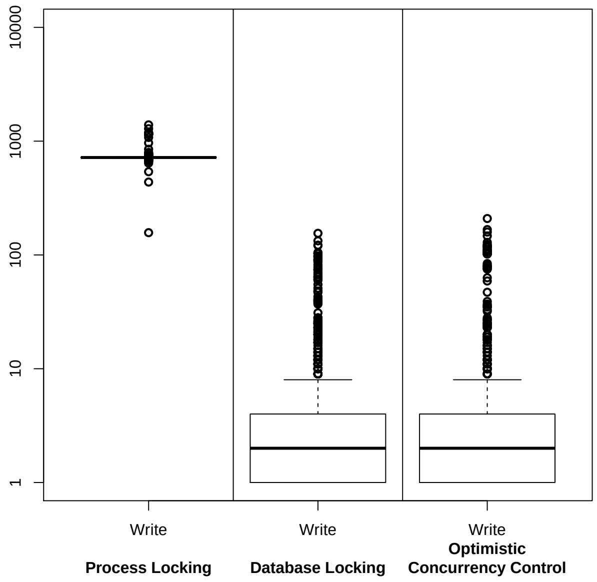


Figure 5.3: Waiting Time for CC3: Parallel Writing, Different Databases

5.1.4.1 Queries

1. Write to database
   - Updating: Yes
   - Frequency: High
   - Runtime: Medium
   - Locking: Varying single database, not known in advance

5.1.4.2 Benchmark

As the databases to be locked are not known in advance, conservative two phase locking degrades to the performance of process locking in Figure 5.4. For optimistic concurrency control, the access pattern does not differ from CC3, it keeps the same low waiting time.

5.2 Case Studies

5.2.1 CS1: Web Application

This case study simulates a web application handling lots of parallel requests, all reading a common configuration database and another random one out of a set of 100 databases

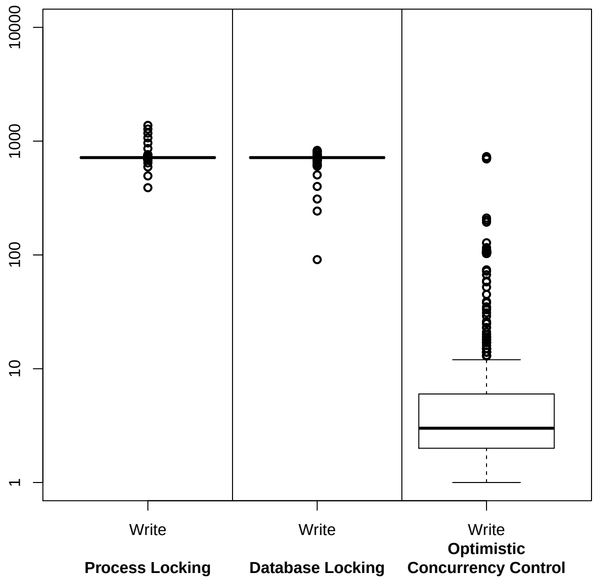


Figure 5.4: Waiting Time for CC4: Parallel Writing, Different Databases, Unkown at Compile Time

caching user information. A small percentage of requests updates the user database involving an HTTP `GET` operation.

5.2.1.1 Queries

1. Read from cache
   - Updating: No
   - Frequency: High
   - Runtime: Short
   - Locking: Configuration and a single cache database, known in advance
2. Update cache from HTTP resource
   - Updating: Yes
   - Frequency: Rare
   - Runtime: Medium
   - Locking: A single cache database, known in advance

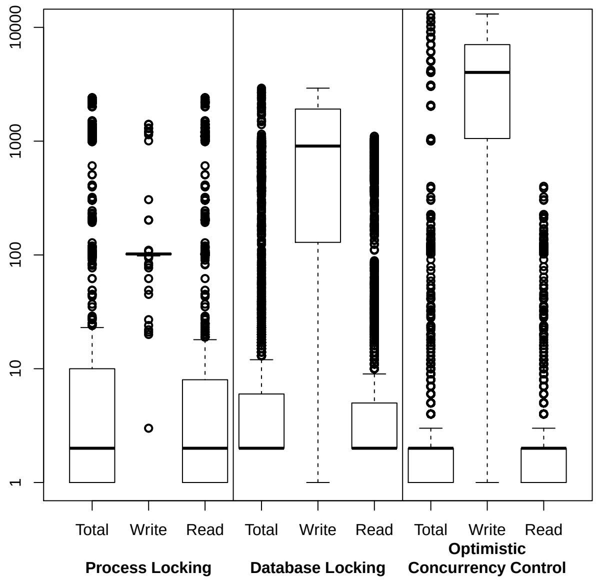


Figure 5.5: Waiting Time for CS1: Web Application

5.2.1.2 Benchmark

Surprisingly, Figure 5.5 shows heavily degrading write performance. This is to be explained by starvation, while process locking strictly runs queries in their arrival order (and thus stops new readers from running when a writer is waiting). Database locking does not, which shows up as a drawback in this use case. Overall performance is slightly better, but the spread increased significantly.

Optimistic concurrency control suffers even further, as of two parallel writers to the same database only the first to commit may pass the validation phase; the other has to restart and his lost calculation time gets added to his waiting time. Anyway, read and overall performance are slightly better than both process and database locking ones.

5.2.2 CS2: Web Shop Backend

This more complex example consists of a majority of small read-only transactions (fetch product information). Some less frequent operations like orders and order management require complex queries with medium runtime and undeterminable databases prior to compilation.

5.2.2.1 Queries

1. Fetch product information
   - Updating: No
   - Frequency: High
   - Runtime: Short
   - Locking: Single database, known in advance
2. Order management
   - Updating: Yes
   - Frequency: Rare
   - Runtime: Medium
   - Locking: Product and order databases, unknown in advance

5.2.2.2 Benchmark

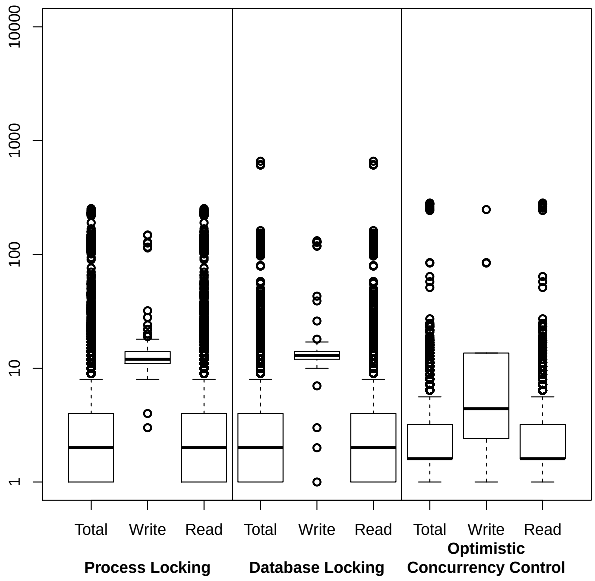


Figure 5.6: Waiting Time for CS2: Web Application

As databases to be locked cannot be determined in advance, process and database locking plots mostly resemble each other in Figure 5.6. Optimistic concurrency control again benefits regarding the updating queries, but is also faster for read-only-transactions: As they're very short, chances are high they can commit while writing transactions still remain in their read-phase and collect changes in their pending update list.

6 Related Work

All scientific related work found during research for this thesis worked on improving concurrency at node level using tree concurrency algorithms, which was discovered being inadequate for the sequential XML encoding used by BaseX. This chapter regards other native XML database systems and how they implement concurrency control.

6.1 MarkLogic 6

MarkLogic[1] uses strict two phase locking on document level [Hunter 2013:35]. Locks are acquired during the execution of a transaction, so deadlock recognition is required. Multi version concurrency control is available and can be enabled if desired. It offers transactions spanning multiple queries [MarkLogic Corporation 2013].

6.2 Sedna

Sedna[2] is a native XML database developed as open source software at the Russian Acadamy of Sciences. It uses a schema-based clustering storage [Haustein 2005:30] that could support tree locking protocols.

Sedna's locking strategy is similar to MarkLogic's, but there are differences in versioning and transaction management. Multiple versions are stored to process read-only transactions while others are writing, but it only keeps a limited number and does not fully implement MVCC. Transactions can span multiple queries, but recovery is limited for long-running read-only transactions [Kalinin and Zavaritsky].

6.3 eXist DB

eXist DB[3] stores XML documents in k-way-trees [Haustein 2005:29], which would support fine granular locking on node level using a fitted tree concurrency control protocol.

[1] http://www.marklogic.com/.

[2] http://www.sedna.org/.

[3] http://www.exist-db.org/.

ACID transactions are not yet fully supported.[4] Read-only transactions can bypass concurrency control. Versioning is offered on document-level, but needs to be triggered manually.

[4] http://exist-db.org/exist/apps/doc/roadmap.xml.

7 Conclusion

When regarding XML encoding schemes, very fine granular tree locking algorithms showed up as not adequate for the sequential XML encoding used by BaseX. A decision was made on implementing conservative strict two phase locking to enhance concurrency control in this native XML database. Conservativeness was chosen to support all kinds of queries, including transactions that may not be restarted because of uncontrollable, external side effects. It was expected this will only be a starting point which will be improved as needed.

Performance tests against an implementation of optimistic concurrency control clearly showed that extending with a non-conservative variation would be desirable. Further, starvation can be a problem that should be dealt with.

7.1 Future Work

There are two major ways to further improve concurrency control,

1. by introducing restartable transactions and
2. by keeping multiple versions.

7.1.1 Restartable Transactions

Restarting transactions are required for dealing with deadlocks that can occur in non-conservative two phase locking, and they're needed for non-locking schedulers like optimistic concurrency control.

To offer those, it must be ensured that no transactions get aborted during their write phase without possibility to roll-back the changes already applied. Luckily, with XQuery Update there is no need to roll back writes as strict two phase locking is enforced by the pending update list and prevents cascading rollbacks (see the section on two phase locking).

When transactions are restarted, they should await the transaction restarting them to commit, as chances are high they will fail again.

Restartable transactions would also allow BaseX to roll-back a transaction after external failures and restart it gracefully, making it easier to embed it in a complex environment.

It is possible to run both restartable and “conservative” transactions at the same time. Conservative transactions could be determined by searching the abstract syntax tree for expressions involving external side effects; or by having the database user decide on this who has more knowledge on whether restarts are acceptable or not.

7.1.2 Keeping Multiple Versions

Multi version concurrency control would not only enhance parallel execution by always allowing read-only transactions to run without waiting for writing transactions to finish, but also add robustness against both hardware and software failures and also power outages leading to inconsistent state on secondary storage, as there always exists an untampered version that reflects the state of the last committed transaction.

Keeping old snapshots from being garbage-collected offers cheap incremental backups without the need of copying the whole database.

For implementing multi version concurrency control, a notably amount of preliminary work has to be completed as most of the data structures are not build on pages yet. A common page controller for both the main nodes table and indices should be considered, as it reduces the complexity of managing and synchronizing multiple versions.

Appendix

Bibliography

Chamberlin, D., Robie, J., Florescu, D., Boag, S., Siméon, J., and Fernández, M.F. 2007. XQuery 1.0: An XML Query Language. .

Chamberlin, D., Robie, J., Florescu, D., Melton, J., Siméon, J., and Dyck, M. 2009. XQuery Update Facility 1.0. .

Grust, T. 2002. Accelerating XPath location steps. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ACM, 109–120.

Grün, C. 2010. Storing and querying large XML instances. .

Haerder, T. and Reuter, A. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4, 287–317.

Haustein, M. and Härder, T. 2003. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. In: L. Kalinichenko, R. Manthey, B. Thalheim and U. Wloka, eds., *Advances in Databases and Information Systems.* Springer Berlin Heidelberg, 88–102.

Haustein, M. and Härder, T. 2004. Adjustable Transaction Isolation in XML Database Management Systems. In: Z. Bellahsène, T. Milo, M. Rys, D. Suciu and R. Unland, eds., *Database and XML Technologies.* Springer Berlin Heidelberg, 173–188.

Haustein, M.P. 2005. *Feingranulare Transaktionsisolation in nativen XML-Datenbanksystemen.* Verlag Dr. Hut.

Helmer, S., Kanne, C.-C., and Moerkotte, G. 2001. *Isolation in XML Bases.* .

Helmer, S., Kanne, C.-C., and Moerkotte, G. 2004. Evaluating Lock-based Protocols for Cooperation on XML Documents. *SIGMOD RECORD 33*, 58–63.

Hunter, J. 2013. *Inside MarkLogic Server.* .

Kalinin, A. and Zavaritsky, N. Sedna XML DBMS: Transactions and Recovery. http://de.slideshare.net/shcheklein/sedna-xml-datbase-transactions-and-recovery.

Kircher, L. 2010. BaseX: Extending a Native XML Database with XQuery Update. http://nbn-resolving.de/urn:nbn:de:bsz:352-154882.

Kung, H.T. and Robinson, J.T. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2, 213–226.

MarkLogic Corporation. 2013. *Understanding Transactions in MarkLogic Server.* MarkLogic Corporation.

O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., and Westbury, N. 2004. ORDPATHs: insert-friendly XML node labels. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, ACM, 903–908.

Retter, A. 2012. *RESTXQ 1.0: RESTful Annotations for XQuery 3.0.* .

Weikum, G. and Vossen, G. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery (The Morgan Kaufmann Series in Data Management Systems).* Morgan Kaufmann.

Weiler, A. 2010. Client-/Server-Architektur in XML Datenbanken. http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-123668.