Universität Konstanz
Department of Computer and Information Science

Bachelor Thesis for the degree

Bachelor of Science (B.Sc.) in Information Engineering

BaseX: Extending a native XML database with XQuery Update

by
Lukas Kircher
(646244)

1st Referee: Prof. Dr. Marc H. Scholl
2nd Referee: Prof. Dr. Marcel Waldvogel

Advisor: Christian Grün

Konstanz, September 26th, 2010

Acknowledgements

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

Special thanks go to Prof. Dr. Marc H. Scholl and Prof. Dr. Marcel Waldvogel for being my referees and taking the time to read this somewhat lengthy document.

Foremost I would like to express my gratitude to my advisor Christian Grün, who supported me throughout the whole process of my studies. As the father of BaseX (which is awesome by the way) and an early mentor of mine, I'm positive he has taught me a thing or two or three hundred.

Last but not least, I want to thank Alex, Andi, Elmedin and Michael (the guys who share office with me) for being good company!

Abstract

This Bachelor Thesis describes concepts behind the extension of a native XML database with XQuery Update. BaseX is a compact and highly efficient open source XML database and XQuery processor. Based on a relational document encoding, a generic method is presented, that exploits the sequential encoding scheme to its full advantage. The implementation yields excellent test results. Further optimizations are proposed. A method to speed up the execution of critical structural updates and ways to accelerate the overall process. A discussion on ACID conformity in transactional XML database systems finalizes this work.

Zusammenfassung

Diese Bachelor-Arbeit beschreibt die Konzepte hinter der Erweiterung einer nativen XML-Datenbank um XQuery Update. BaseX ist eine kompakte und hoch-effiziente open-source XML-Datenbank und XQuery Prozessor. Basierend auf einer relationalen Dokumenten-Kodierung wird eine generische Methode vorgestellt, welche das sequentielle Kodierungsschema zu ihren Vorteilen nutzt. Die Implementierung liefert exzellente Testergebnisse. Weitere Optimierungen werden vorgeschlagen. Eine Methode, um die Ausführung von kritischen strukturellen Änderungen zu beschleunigen, und weitere, die den allgemeinen Prozess verkuerzen. Eine Diskussion bezüglich ACID-Konformität in transaktionalen Datenbanksystemen schließt die Arbeit ab.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Overview 2

2 Preliminaries **3**

2.1 XML 3

2.2 XML Databases 3

2.3 XML Encoding Schemes 4

2.4 BaseX 6

2.4.1 Table Encoding in BaseX 6

3 XQuery Update **9**

3.1 XQuery Update 9

3.2 New Expressions 10

3.2.1 Insert 10

3.2.2 Delete 11

3.2.3 Replace 12

3.2.4 Rename 12

3.2.5 Transform 12

3.3 Pending Update List 13

3.4 Checking data model constraints 14

3.5 Order of Updates 14

4 Implementation **16**

4.1 Overview 16

4.2 *Pre* values vs. node *IDs* as identifiers 16

4.3 Architecture 18

4.4 Applying Updates 19

4.4.1 Checking Constraints 20

4.4.2 Execution Example 21

4.5 Insert Before Statement 22

4.6 Text Node Merging 23

4.7 Fragment Processing 24

5 Performance **26**

5.1 Test settings 26

5.2 MonetDB 27

5.3 Test Queries 28

5.4 Results 30

5.5 Observations 30

6 Future Work **36**

6.1 Overview 36

6.2 Optimizations 37

6.2.1 Fragmentation 37

6.2.2 Optimizing the Pending Update List 37

6.2.3 Block-wise replace 38

6.2.4 Page-wise updates 39

6.3 Rollbacks and Recovery 42

7 Conclusion **45**

Bibliography **47**

Chapter 1

Introduction

1.1 Motivation

Compared to a dinosaur like the field of relational database systems, the world of XML is still a young one. Emerging in almost every field imaginable, XML is not only used in large database applications, but on a smaller scale as well. Having emerged as the most versatile file format for data exchange of all kinds and in all aspects of life, the need for, not only further developed, but also more specialized database systems is growing steadily.

Yet, XML not exactly had its deserved break-through. Well known commercial database providers like Oracle, IBM and Microsoft extended their large scale solutions with the support for XML already years ago. Still, slightly outdated prejudices exist. If we look for something to blame this on, we might make a find with the late introduction of a standardized method to manipulate XML data. Database technology without a well documented and widely accepted updating standard was considered outdated even years ago.

Several attempts have been made [1] [2]. With the release of the XQuery Update Facility candidate recommendation (XQUF), the quest has hopefully come to an end. As the specification is still young, trying to establish BaseX [6] among its early adopters seemed worthwhile.

Over the last few years, BaseX has been steadily approaching the state of a fully developed database system. With the possibility of manipulating

[1] http://www.w3.org/TR/xslt20/, *access: 2010-09-21*

[2] http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html, *access: 2010-09-21*

databases but lacking an interface to actually perform these, adding support for the XQUF bubbled itself to the top of our to-do list. Integrating an update standard rises a lot of questions. What is generally possible? How do we deal with concurrency? What can be done to prevent data loss? How far can we take it performance-wise?

BaseX features a compact and highly optimized back end. With a sequential table encoding tailored to excel at querying, we are eager to find out how far we can take it with updates. Adding an updating mechanism requires far-reaching changes and an architecture that is adaptable to future needs. At the same time overhead is to be kept at bay. The updating module should exhibit the same qualities as the rest of the system: compactness and efficiency.

To be competitive, a database system has to measure itself with the ACID properties [9] - a respected set of criteria that describe essential qualities for transactional database systems. Identifying insufficiencies and ways to solve them is another question to be answered.

1.2 Overview

Chapter 2 introduces some basic concepts of a native XML database system. Specialties of XQuery Update and an overview of the new available functions follow afterwards (Chapter 3). The implementation process of this work together with problems, that had to be solved, is documented in Chapter 4. Related work and other projects have to compete against our implementation in Chapter 5.

The implementation of XQuery Update itself opened up a multitude of possible directions for future work. To help BaseX become a reliable and mature XML database, some non-trivial database mechanisms are yet to be added. Features like recovery may be well established within relational database systems. Due to the hierarchical nature of XML data and the compact storage of BaseX, the challenges are of a different kind. Some problems still wait to be solved a better way. A glimpse on this together with suggestions to improve overall performance is given in Chapter 6. Finally, a conclusion can be found in Chapter 7.

Chapter 2

Preliminaries

2.1 XML

XML is great.

As I couldn't do it any better, an introduction featuring all its bells and whistles is left to [3, 6, 7].

2.2 XML Databases

The growing popularity of XML as a storage and exchange format leads to new challenges. The result is an increase in number and size of XML documents waiting to be processed. Providing efficient and reliable methods for storing and processing XML are therefore of substantial importance. Traditional database technology has been on the market for a long time. As the requirements to a database are the same, for relational data and XML data, making use of this highly developed technology seems a good start. Yet, the specialties of XML and its structured data model complicate the issue. The XML:DB [1] initiative divides XML databases into three categories:

- **XML enabled databases** build upon existing databases. Oracle uses this approach for example. Exploiting well established database technology brings several advantages. These systems have been tested extensively in regards to huge data spaces and performance. Advanced

[1] http://xmldb-org.sourceforge.net/faqs.html, *date of access: 2010-09-18*

database features are already included. XML enabled databases rely on relational- or object-oriented data models and an XML mapping on top. As these databases are not specifically designed for XML, data can be accessed in various ways (SQL, XPATH ...). Composing and decomposing an XML document involves additional steps.

- **Native XML databases** are built from scratch to process XML data. This kind of database defines a model for XML data and operates directly on this model. An XML document serves as the fundamental storage unit. The physical storage model can be chosen freely. No additional steps are required for composing and decomposing XML.
- **Hybrid XML databases** aggregate the properties of the other two types.

2.3 XML Encoding Schemes

Finding a suitable document representation is key for efficient query processing. Mapping documents to a relational encoding is one way to do this. XML data is basically a tree. As trees are a well known structure, a variety of possible conversions exist. Concerning XML, the particular and minimal requirements to such a mapping are [5, 2.3,2.4]:

- A unique node *id* must be assigned to each node
- *Document order*, which is equivalent to the result of a preorder traversal, must be preserved

In addition, fast evaluation of all XPath axes and efficient support of updates complete the list of requirements. In general, two different approaches exist. While a sequential encoding does not represent the tree structure by nature, ids assigned by an hierarchical approach include information on node relationships and document order.

[10] presents a hierarchical encoding scheme that works especially well for update operations. A node is identified by a variable length key like 1.3.3. A byte-by-byte comparison of these keys yields the document order and hierarchy. Exclusively using odd numbers during the initial mapping leaves

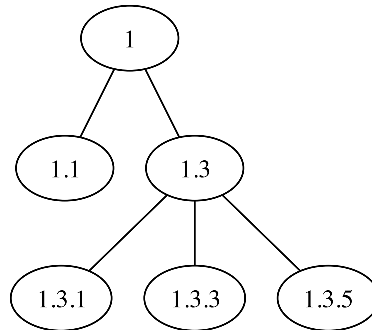


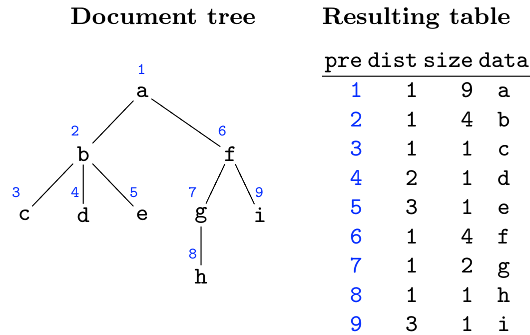
Figure 2.1: ORDPATH key labelling

room for the insertion of subtrees at a later point. For example, if new siblings are added between two existing siblings 1.3.1 and 1.3.3, the new keys are extended with an additional level using even numbers. In this example the inserted nodes end up with the keys 1.3.2.1, 1.3.2.3 etc., for example. Even numbers only influence the document order. They have no effect on the depth of a node or on any ancestor- and descendant- relationships.

Using the ORDPATH encoding scheme, an updating process omits the usually necessary relabeling of previously existing nodes. However, node keys, being highly structure dependent and of variable length, need to be stored in an efficient manner to overcome their verbose nature.

Using a sequential encoding scheme, the document nodes are labeled sequentially in document order. As a result, node ids do not contain any information on the hierarchical structure, but, due to their fixed length, are a great start to achieve a compact storage layout.

Grust proposes such a distinct scheme that provides efficient evaluation of all XPath axes as well as a compact storage representation [8]. The core of his work is the *pre/post* plane. The document nodes are assigned a *pre* and *post* value, according to their positions they take during a pre-order and post-order tree traversal. Each document node induces a partition of this plane into four disjoint regions, representing the major XPath axes *ancestor*, *descendant*, *preceding* and *following*. Evaluating one of these axes finally comes down to a region query, a kind of query that is highly optimizable with the help of additional tree structures. In order to grant full access to all XPath



pre	dist	size	data
1	1	9	a
2	1	4	b
3	1	1	c
4	2	1	d
5	3	1	e
6	1	4	f
7	1	2	g
8	1	1	h
9	3	1	i

Figure 2.2: Table Encoding in BaseX

axes, Grust adds parent, attribute and tag information to each node. The result is a five-dimensional descriptor, that is created for each node during the initial parsing of a document. As said before, using a sequential approach helps with realizing a compact storage layout, mapping the properties of a node to fixed length keys. Yet, as the *pre* and *post* values represent the document order and hierarchy, performing updates is expensive. Facing the worst-case scenario, updating can lead to a relabeling of all document nodes.

2.4 BaseX

A native XML database using a sequential encoding approach is the open source project BaseX. BaseX is developed by the Database and Information Systems group at the University of Konstanz, led by Professor Dr. Marc H. Scholl. With Christian Grün as the main developer, today it is renowned as a fast and versatile XQuery processor and XML database. Offering a complete implementation of W3C's XQuery 1.0 and its Full Text extension, XQuery Update emerged as the most desirable feature on the never-ending to-do list.

2.4.1 Table Encoding in BaseX

BaseX makes several adjustments to the encoding scheme to speed up query evaluation and optimize memory consumption. An example of the used encoding scheme is given in figure 2.2.

In addition to the *pre* value, which also serves as a node id, BaseX uses *distance* and *size* values to keep track of the node relations in a document.

This is due to several reasons:

1. In contrast to the absolute *parent* value, the *distance* gives information on the number of nodes lying between a parent and child node. Using an absolute parent value, all child nodes have to be updated if the parent changes its *pre* value. We keep in mind: $parent = pre - distance$. In conclusion, the *distance* value supports a fast evaluation of the *ancestor* axis without raising costs.

2. The *size* value provides information on the number of descendant nodes and therefore fast access to the following siblings.

As the value of a node (name, attribute value, text etc.) is not stored in the table but referenced, all information on a node can be stored occupying constant memory. Together with the *pre* value, a node on disk can be accessed in constant time via calculating the offset. All kinds of nodes are stored in the same table (including text and attribute nodes) where a *kind* value (not shown) helps with distinction and helps realizing the full axis feature.

Yet, this encoding scheme also suffers from the usual drawbacks regarding update operations. A carefully chosen table layout minimizes the negative effects. As explained above, using relative values wherever possible (*distance*, *size*) reduces the amount of necessary node updates. Let's assume we add a subtree t as an only child to an element e, with s being the number of nodes in t. Hence, the following values have to be updated:

1. The *pre* values of all nodes on the following-axis of e are increased by s.

2. The *size* values of e and all its ancestors are increased by s.

3. All *distance* values of the nodes on the following-sibling axis of e are increased by s.

This applies accordingly to a delete operation.

Of course, changing the *pre* values after an update makes them essentially useless as node identifiers. A unique node id is therefore assigned to each node that does not change during a node life cycle. To keep track of nodes,

a mapping between *pre* values and node ids is used, that is of substantial meaning in scope of this work. This topic is discussed in detail at a later point.

Explaining experimental results in a later chapter requires some deeper knowledge of the storage design. As mentioned before, if documents are stored in a sequential manner on disk, certain update operations will lead to a substantial slowdown due to tuple shifting. For example, if a node at the lowest possible *pre* value is deleted or inserted, every node below would have to be shifted on disk by the number of deleted or inserted nodes.

To keep costs low, BaseX stores its database files not in one location, but divides a table into pages. The file system manager subsequently decides where these pages are stored on disk. A directory, which resides in main memory, stores the first *pre* value together with its page reference. Each page initially contains 256 tuples, and knowing the lowest *pre* value a following page contains, makes it simple to find the exact location of a node. Using this approach, making structural changes no longer results in shifting half of the document on average. These drawbacks are limited to a single page for the deletion of a node of size one. As gaps are not allowed within pages, deleting or inserting a single node would result in $(pageentries) - 1$ shifts, worst case. Gaps are shifted to the end of a page. Further details can be looked up in [6].

Yet frequent updates lead to fragmented and growing database files which could result in poor query performance and storage efficiency. A solution for this is proposed in Chapter 5.

Chapter 3

XQuery Update

3.1 XQuery Update

There are several extensions available for XQuery that add further functionality to the language and with this give the possibility for a much broader application. The Full Text extension specifies expressions and functionality to retrieve tokens in data sets that contain full text, with BaseX being among the first systems to support this standard [1,7]. Another extension is XQuery Update [4]. Besides the base language that provides no functionality to manipulate XML data, the XQuery Update Facility (XQUF) basically enables us to do the following:

1. Insert nodes into an existing document.
2. Delete nodes from an existing document.
3. Modify nodes while preserving their identity.
4. Modify copies of existing nodes via the *transform* expression. The original node identities remain untouched.
5. Serialize an XDM instance to secondary storage via *fn:put*.

The five new kinds of expressions, added by XQuery Update, are *rename*, *insert*, *delete*, *replace* and *transform*. *Rename*, *insert*, *delete* and *replace* expressions are specified as basic updating expressions or updating expressions.

Every expression that is not an updating expression is categorized as a simple expression. This applies to the *transform* expression as well. Within the scope of a *transform* expression nodes to be modified are copied in advance. As a consequence, the actual target node identities remain untouched. A function called *put* provides functionality to serialize XDM instances to secondary storage. Using this function, serialized documents can be subsequently accessed using the *fn:doc* function.

3.2 New Expressions

Using XQuery Update is pretty straightforward if we keep a few things in mind. The following part introduces some basic knowledge and terminology to simplify explanations at a later point. The specification provides an in-depth view on the update facility. So first are examples for all new expressions categorized as updating expression (l.a.), as well as the available options. Regarding the *insert* expression this is especially important as issues arise with implementation. Afterwards I will give a few details about the new non-updating expression, namely *transform*, and the *put* function.

In XQUF, the keywords 'node' and 'nodes' can be used interchangeably without any consequences.

3.2.1 Insert

```
insert node (attribute {'a'}{5}, 'text', <e/>)
as first into /n
```

Nodes are inserted at the given location. The location basically consists of a single target node and a modifier to further specify the insert location with respect to this target. Available options are *into*, *into as first*, *into as last*, *before* and *after*, see figure 3.1.

Using the *into* modifier, the insert position on the child-axis of the target node is implementation dependent. However, it may not interfere with other modifiers that define an exact location in relation to a target. In scope of this work, *'insert into'* adds nodes after the last position on the child-axis. If

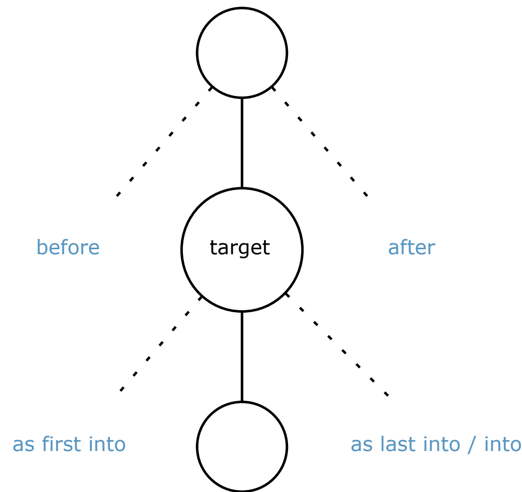


Figure 3.1: location modifiers for insert expression

another *insert* statement on the same target uses the *'into as last'* modifier, the final node order must reflect this.

If a distinct combination of a modifier and a target node is used several times during a snapshot, the sequence order among the inserted nodes is implementation depended. This also applies to the *replace* expression. Of course, XDM constraints may not be hurt.

A document node is not inserted itself, but is replaced by its children.

3.2.2 Delete

```
delete node //country
```

All *country* elements are deleted in the example. In contrast to other updating expressions where multiple targets are not allowed, the target expression may return a set of nodes. Therefore it is not necessary to use an iterative approach, like a FLOWR expression, to delete multiple nodes. Within a query, it is possible to rename and delete the same node for example. The outcome and concepts behind this are given below.

3.2.3 Replace

There are several ways to use a *replace* expression. Used without the *'value of'* option, the target itself is replaced by the insertion sequence and its identity is lost.

```
replace value of node /n
  with (attribute {'a'} {5},<a/>, 'xx')
```

The <n> element is replaced with the order of the insertion sequence being crucial. For example, attribute nodes are not allowed to succeed element nodes in the insertion sequence, as they precede them on the descendent axis of the target. This leads to additional challenges during implementation and applies accordingly to every other expression where insertion sequences are used.

Another option is to replace the value of a node with a string value and preserve its identity. If the target node is an element, all children are deleted and replaced with the given text node.

3.2.4 Rename

```
for $n in //n
  return (
    rename node $n as 'newN',
    rename node $n/@id as 'newId'
  )
```

A *rename* expression replaces the name of a node with the given name. Nodes on the attribute- or descendant-axis of the target are not affected. If required, this has to be done explicitly.

3.2.5 Transform

```
copy $c := doc('/myDB/doc.xml')//n
modify rename node $c as 'copyOfN'
return $c
```

All <n> elements in the document are copied and subsequently renamed. As updates are performed on node copies, the document remains untouched. *Transform* is the only way to make changes that are visible during a snapshot. The updated nodes are directly returned and can be passed on to other expressions or functions.

3.3 Pending Update List

I will introduce the concept of the pending update list (PUL) with an example. Consider we perform a query on the following document:

```
doc.xml:
  <doc> <a/> </doc>

query:
  insert node <b/> into /doc,
  for $n in /doc/child::node()
    return rename node $n as 'c'

doc.xml:
  <doc> <c/><b/> </doc
```

The result may be unexpected but reveals instantly the core concept of XQUF. The former <a> element is the only node to be renamed, despite the flwor-expression iteratively renaming all children of <doc>.

XQUF adds the updating expression as a new category of expression to the processing model. *Insert-*, *delete-*, *replace-* and *rename-* expressions represent this category, as they allow altering the state of a node. With XQuery Update, an expression now returns a pending update list (PUL) together with an XDM instance. This PUL consists of update primitives, with each primitive representing a single node state change, triggered by the expression. Prior to execution, all individual lists are merged together forming a query-wise global PUL. The node state changes on this list are held pending until the end of a query.

As a result, updates are only visible after a snapshot is finished. Within a query it is not possible to access any changed node states, with the *transform*

expression forming a single exception. On one side, this greatly simplifies the implementation process, eliminating issues, regarding concurrent update operations, before they even arise. On the other, the query-writing process can be challenging at first.

3.4 Checking data model constraints

Before node state changes are applied, all primitives are checked for compatibility. This concept follows the atomicity (ACID) principle. If conflicts are detected, the update process is aborted and databases are left unchanged. Compatibility tests are applied at any time during a query, no matter if they belong to the static or dynamic context. Of course, finding errors as soon as possible leads to shorter processing times. More details can be found in Chapter 4, as a lot of information on this is implementation dependent.

3.5 Order of Updates

After all update primitives are gathered on a global PUL we have an unordered set of operations without a specific one to begin with. To support the concept of a global PUL, the specification suggests an actual order for all update processes. The idea is to create a hierarchy and by this a set of rules that defines the effect on a node being target of multiple operations. The order is visualized in figure 3.2.

Delete apparently is the last update operation to be executed. If a target node is affected by a *renaming* expression and a *delete* expression, the renaming process would be without effect in the end.

Following this order is a recommendation, however, it simplifies the implementation process and gives room for optimizations by skipping unnecessary operations in the first place. Before *fn:put* is applied, all other operations are finished. Ultimately, nodes that are serialized by *fn:put* reflect all changes made effective during a query. Trying to serialize replaced or deleted node identities results in an exception.

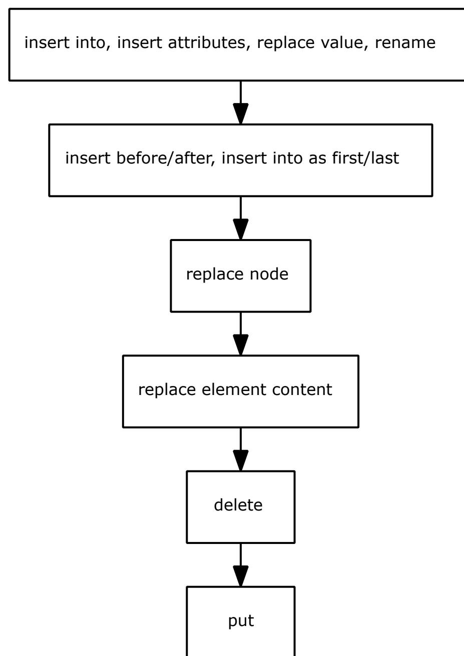


Figure 3.2: order of updates

Chapter 4

Implementation

4.1 Overview

As shown before, BaseX uses a sequential document encoding scheme. Updates consequently result in changing a relational tuple. BaseX already featured atomic update operations like insert, delete and rename prior to the implementation of XQuery Update to make changes to existing tuples. In the scope of this work, the main tasks are:

1. Introducing new expressions to the query parser.
2. Designing an architecture that meets the facility requirements.
3. Implementing this architecture with an eye on the compact and efficient storage layout of BaseX, minimizing overhead wherever possible.

4.2 *Pre* values vs. node *IDs* as identifiers

There are two possibilities of identifying nodes during the query process. The *pre* value is the position of a node during a tree traversal in document order and is guaranteed to be unique within a single document. Using this one, issues arise when several different documents are handled in a query, as duplicates exist.

We could alternatively make use of a unique node id that each node is assigned during document parsing, which is, at this point, equal to its *pre*

number. As this id does not change during a node life-cycle, a mapping between *id* and *pre* values is used that enables us to identify and access individual nodes at any time. Implementing the concept of pending updates would then basically be a no-brainer. However, the mapping becomes invalid if structural updates are performed on a database. Efficient on-the-fly updating of this index is not yet an established feature in BaseX, and has to be called explicitly. Performing updates on the basis of node id's is therefore not an option at the moment. We would also throw the benefits of our sequential document encoding out the window.

As *pre* values represent document order, structural updates (insert, delete) cause a shift of nodes with a *pre* number higher than the updated position. So in case of an update, generally half of the document nodes are to be shifted on disk. But, together with pending updates, we can use this fact to our advantage. The XQUF defines, that each expression returns an XDM instance together with a pending update list (PUL), which accumulates all update operations during a snapshot. Knowing what nodes are to be changed and how, helps us optimizing the process like this:

1. Create an individual PUL for each document that is accessed during a snapshot.
2. Check data model constraints for each document using algorithms tailored to our sequential table encoding.
3. Sort all PULs by the *pre* number of their target nodes and apply bottom to top. Ultimately the *pre* value of target nodes that have not yet been processed are unchanged.

We see, using *ids* instead of *pre* numbers would cost us the benefits of our compact table encoding.

Basically with the usage of *pre* values we can omit to keep *id-pre* mapping up to date. This is the most significant advantage of this approach as BaseX currently lacks an efficient mapping.

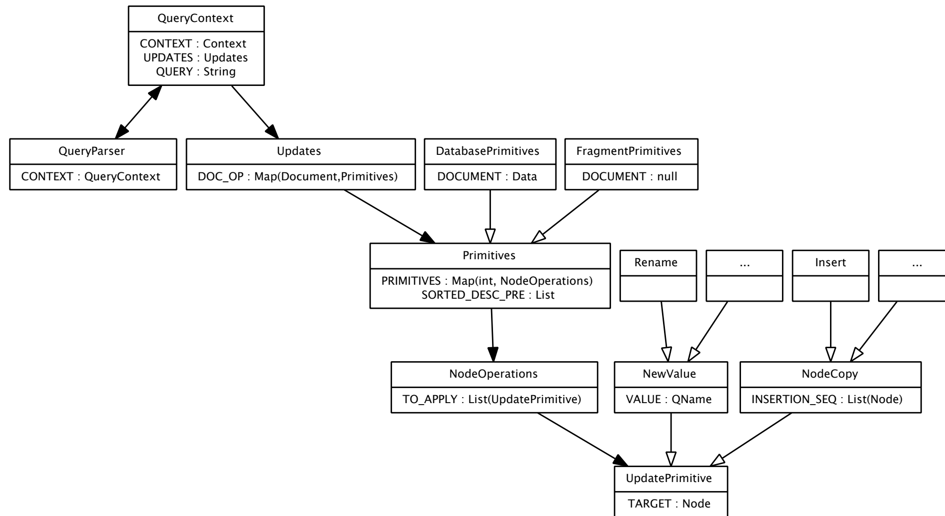


Figure 4.1: architecture of XQuery Update module

4.3 Architecture

As updates are held pending, we need a structure that allows to cache all update primitives. In addition, it is vital for this structure to meet our previously defined requirements as follows.

1. Using the *pre* number of a node for identification during a snapshot.
2. Caching update primitives separately for each accessed document to handle *pre* value duplicates among target nodes and to finally apply updates from bottom to top like stated above.
3. Using processing algorithms tailored to our sequential document encoding.
4. Caching insertion sequences for *insert*- and *replace*- statements.
5. Supporting the recommended order of update operations.
6. Leaving enough room for optimizations.

Figure 4.1 displays the results of our study. Update primitives are extracted by the query parser *(QueryParser)* and passed on to the query context of

BaseX *(QueryContext)*, which holds all relevant context for a query, like accessed documents, variables, functions and more. Update operations triggered by XQuery Update are managed by this class as well. The *Update* class stores individual update primitives document-wise, keeping a mapping between each accessed document and a collection of update operations targeting this document *(Primitives)*. *Primitives* subsequently maps *pre* values of target nodes to their corresponding update primitives *(NodeOperations)*. Not only enables this to sort updates by *pre* values prior to execution and take advantage of our table encoding. But it speeds up the allocation process of newly parsed updates as well, using hashing structures with constant access time. Finally *NodeOperations* gathers update statements *(UpdatePrimitive)* for a single target node, conforming to the recommended update primitive order. Several types of update primitives are modeled, depending on the operation. Structural updates adding new nodes to a database (insert,replace) are represented by the *NodeCopy* class which caches source nodes before any updates are applied. Value updates like the *rename* expression are covered by *NewValue*. Similar to *NodeCopy*, the new QName attribute is cached for later application.

Keeping temporary copies is necessary to prevent dirty reads and accessing wrong or no longer existing nodes - which is a constant issue as *pre* values change with structural updates. *Primitives* are further divided into database and fragment primitives, since we treat fragment primitives differently.

4.4 Applying Updates

After all updates are gathered on a global pending update list they are executed document-wise. Before any changes are applied, compatibility tests either ensure a positive outcome of a query or yield an error code and abort the process.

In principle the execution of a query is divided into three stages.

1. Parsing of a query and detection of static errors.
2. Testing compatibility of primitives for each database to ensure atomicity (ACID).

3. Applying updates and propagate changes to persistent storage. Serialize documents created with *fn:put*.

Additional information on step one is beyond the scope of this work, but further details can be looked up in [6].

4.4.1 Checking Constraints

Some tests can only be performed after our global PUL is complete. XDM constraints may be hurt at times during the query process. However, after evaluation all accessed documents must be in a valid state. Thereby the following statement is perfectly correct:

```
doc.xml:
  <a id="0"/>

query:
  insert node attribute {"id"} {1} into /a,
  delete node /a/@id
```

As all update primitives are accumulated and held pending, the position of a statement within a query has no effect on the time of execution. The above query may lead to a temporarily duplicate *id* attribute. Still, with the original attribute being deleted, the final result is valid.

Error codes are divided into two categories. Static errors are a result of the parsing process and can be detected at an early stage. Contrarily the detection of dynamic errors is connected to input data. As an update expression changes structure and values of our input documents, some errors of this kind can be found only after all update primitives are collected. These errors include:

- **concurrent operations** Two rename statements on the same target have an undefined outcome.
- **duplicate attributes** See example above.

- **namespace errors** Connecting the same prefix for different nodes with an ancestor- or descendant- relationship to different URIs.
- **applying fn:put on lost node identities** Target nodes and their ancestors may not be deleted, which is detected by traversing the ancestor axis for each root node to be serialized.

Spoken casually, if no one pulls the plug BaseX conforms to the ACID principles.

- **atomicity** At the end of a query either all operations are executed or documents are left unchanged.
- **consistency** All accessed documents are in a consistent state before and after query evaluation.
- **isolation** Among unrelated updating queries one doesn't effect the result of another which is ensured by a scheduler (not scope of this work).
- **durability** Changes on database nodes are persistent.

Of course errors can still occur while updating a database. Things like media failure, operating system- or code-induced errors are common threats for a database system. At this point having no sophisticated recovery or restore mechanisms, BaseX can guarantee atomicity and durability only to a certain extend. Future work regarding these advanced features is discussed in Chapter 6.

4.4.2 Execution Example

To sum up our previous efforts this section provides a simple example for a query evaluation. Figure 4.2 shows the execution tree for the given query:

```
delete node doc('doc1.xml')/a/b/g,
insert node doc('doc1.xml')//n
as first into doc('doc1.xml')/a/b,
replace node doc('doc2.xml')/cc with doc('doc1.xml')/a/n,
rename node doc('doc1.xml')/a/b as 'NewName'
```

The height of the corresponding execution tree is 4. Level 2 contains distinct documents and fragments that are accessed by the query. On the child axis of each document we find database nodes being target of pending updates. Sorting these nodes from high to low *pre* values ensures that updates are applied bottom to top and target *pre* values remain valid. Update primitives are finally appended at leaf-level and apply to the order of updates, stated in figure 3.2.

Execution finally comes down to a pre-order traversal of the tree. Each time we hit a leaf, changes are made effective. The tree is actually traversed twice: On the first pass primitives are tested for compatibility to ensure atomicity and consistency.

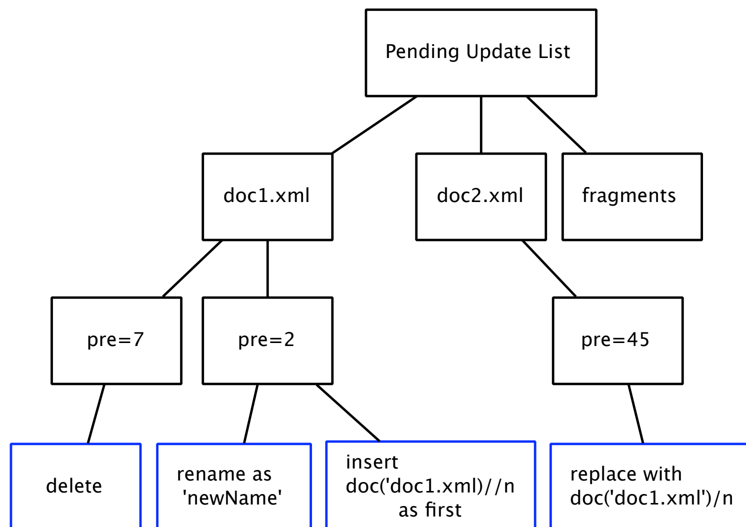


Figure 4.2: execution tree

4.5 Insert Before Statement

Our simple and elegant updating technique, as described above, is in conflict with the *insert before* statement. Using the $before$ modifier in an expression with a target node n, the source nodes are added on the preceding sibling axis of n. The *pre* value of n itself is therefore increased by the number of inserted nodes, as they occupy lower *pre* values. Ultimately, eventual *replace*

or *delete* statements operating on the same node cannot be executed as they follow in the order of updates.

Let s be the number of inserted nodes, n our target and p the target's *pre* value. One simple solution would be to increase p by s for all following update operations on n.

4.6 Text Node Merging

Delete and insert operations may lead to adjacent text nodes. The XQuery data model forbids this. Using our update protocol, detection of adjacent text nodes takes some extra effort. During update execution we cannot perform structural changes on *pre* values lower than the current update position. This would cause a shift of nodes on disk that are ahead on our updating schedule. *Pre* values as node identifiers would be virtually useless as a result. It is also not possible to merge two text nodes with one of them being target of an update primitive not yet applied. For example, if a text node t is inserted after a text node p, these two nodes are adjacent. If t and p, which is also target of a delete primitive, are merged immediately, the value of t is lost with the deletion of p.

We have to make sure merges do not propagate to lower *pre* values. Keen observation reveals that text node adjacency is only present among siblings. Merging nodes parent-wise generates little overhead but solves the issue (4.3).

This also shows the way updates are applied document-wise:

1. Update primitives are applied for each target node of a document, starting with the highest *pre* value *PRE*.
2. *PAR* is the parent of *PRE*. All update primitives that have a target which is a child of *PAR* are applied. *FIRST* holds the lowest *pre* value among the children of *PAR*, which is target of a *'replace'*, *'insert before'*, *'insert after'* or *'delete'*. These update types can lead to text node adjacency that can not be resolved on the fly. *'Insert into'* statements are not a problem, as all other updates on the child axis have already been performed.

3. If such an update has been performed and *FIRST* is therefore greater than *-1*, adjacent text nodes are merged as soon as the current child-axis of *PAR* is left.

As axis changing triggers the merge process, the current solution carries the risk of traversing a distinct child axis multiple times. Yet, being not the general case, this is tolerated as other solutions are memory-consuming.

```
applyUpdates(DOCUMENT)
  PAR <- null
  FIRST <- -1
  for all updates PRE in DOCUMENT:
    NEXTPAR <- parent(PRE)
    if NEXTPAR not PAR:
      if FIRST > -1:
        mergeTexts(PAR, FIRST)
      FIRST <- -1
      PAR <- NEXTPAR
    applyUpdates(PRE)
    FIRST <- structuralUpdateOnSibling(PRE)
  mergeTexts(PAR, FIRST)

mergeTexts(PAR, START)
  if START == 0:
    return
  MAX <- PAR + size(PAR)
  P <- START - 1
  while P < MAX - 1:
    if P is element:
         P += size(P)
    else if P is text and kind(P + 1) is TEXT:
      merge(P, P + 1)
    else:
      P++
```

Figure 4.3: merging text nodes

4.7 Fragment Processing

As updates are not visible during a query, the *renaming* statement has no persistent effect on our document, see figure 4.4.

The XQUF does not distinguish between fragment and database nodes as a target. True to the nature of a fragment, updates during a query are non-

```
let $n := <a><n/></a>
return (
  rename node $n as 'b',
  insert node $n into doc('doc.xml')/n
)

doc.xml:
  <n>
    <a><n/></a>
  <n/>
```

Figure 4.4: updating a fragment

persistent and therefore not visible at all. Adding the fact that compatibility and data model checks cause some overhead it's only consequent to skip the processing of fragment updates altogether. Unfortunately, documents serialized with *fn:put* are an exception to the rule as they reflect the results of a snapshot.

Our implementation of the *transform* expression helps us to fill this gap. We first create a main memory database instance of the nodes in the *copy* statement and apply all updates in the *modify* statement on this instance. The result is immediately visible and can be serialized with *fn:put* or passed on to other expressions. An example is given below.

```
let $n := (copy $t := <a><n/></a>
  modify rename node $t as 'b'
  return $t)
return insert node $n into doc('doc.xml')/n

doc.xml:
  <n>
    <b><n/></b>
  <n/>
```

Of course, this approach is limited as we keep our temporary database in main memory. However, these database instances are expected to be small in general.

Chapter 5

Performance

5.1 Test settings

Goal of this test was to determine the overall performance and scalability of our approach on varying database sizes. The XMark benchmark [11] proposes a combination of an XML document together with an assortment of queries, that test the overall real-life performance of a system. As a lot of implementations adopted this testing strategy during the last years, it seems a good starting point to gain knowledge about update performance as well and to generate comparable results. Queries and documents used are explained in detail below. Documents were created using the XMark generator with scaling factors of 0.001, 0.01, 0.1 and 1.0.

Operating System	Linux-openSuse 11.1 64-Bit
Processor	2x Intel(R) Xeon(R) CPU E5345 @ 2.33GHz, 8 cores
Memory	33GB RAM
Secondary Storage	919.4 GB
Java Version	OpenJDK Runtime Environment 1.6.0

Table 5.1: computing machine specifications

Tests were performed on the given machine 5.1 using the client- and server-architecture of both systems. Execution times for the documents 110KB.xml and 1MB.xml were measured 20 times in a row. As processing took a lot

longer for the files 11MB.xml and 111MB.xml, the number of runs was limited to 10. BaseX was tested with version 6.2.5. For each combination of document size, query and database, the shortest execution time of all runs is listed in 5.3.

On a side note, running the tests with BaseX assigning no additional memory for the Java Virtual Machine (-Xmx flag) did not reveal any differences. Running single threaded, BaseX does not benefit using eight cores instead of one.

Document	**Input Size**	**Nodes**	**#item**	**#date / 4**	**# runs**
110KB.xml	113 KB	3290	22	22	20
1MB.xml	1134 KB	33056	217	252	20
11MB.xml	11396 KB	324274	2175	2324	10
111MB.xml	111 MB	3221926	21750	22545	10

Table 5.2: XMark test document properties

5.2 MonetDB

MonetDB/XQuery [2] is an open source XML database and developed at the Centrum Wiskunde & Informatica, the national research center for mathematics and computer science in Amsterdam, Netherlands. It features the powerful Pathfinder engine, which is introduced in Chapter 2, to speed up query processing. MonetDB has been chosen for its similar storage layout and efficient query processing.

There are certain differences that will later help to explain our test results:

1. Instead of *pre/size/distance* values, MonetDB uses *pre/size/level*. While this is of importance for query performance, it will have a minor effect on the performance of structural updates. In certain cases there is a slight advantage. The *distance* values of following siblings need not to be updated if a subtree is inserted or deleted.

2. In contrast to BaseX, attributes are not in-lined but stored separately. Thus performing structural updates on attributes is expected to be

faster, as they are usually small in number and the main table is not accessed at all.

3. MonetDB differs between read-only and updatable databases. Similar to BaseX, the table is divided into pages holding a fixed number of tuples. These pages may contain gaps in between tuples. Creating an updatable database also gives the possibility of leaving empty space for later insertions on each page. The default configuration (10 % of document size) is used for benchmarking.

4. MonetDB keeps a mapping between *pre* values and node ids realized by holding a view on the original table with pages in logical order.

Queries for MonetDB are altered to support the slightly outdated 'do' syntax and to reference the document which is to be updated. For example, inserting a node in MonetDB looks like this:

```
do insert <test/> as last into doc('xmark')//item/name
```

Calling *'Mclient –version'* returned the following information:

MonetDB server v4.38.5 (64-bit), based on kernel v1.38.5 (64-bit oids) Release Jun2010-SP2

5.3 Test Queries

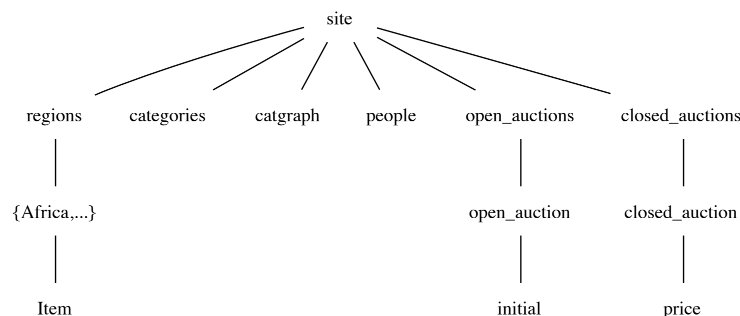


Figure 5.1: structure of XMark documents

Figure 5.1 visualizes part of the structure of all XMark documents. With the exception of the *category*, *people* and *catgraph* elements, all child nodes of the *site* element contain *date* elements on their descendant axis.

To get a general feeling for updating performance, the chosen test queries fall into several categories:

- **Structural update at a single location.** The execution time for single updates is taken at the beginning, in the middle and at the end of a document *(Q1, Q2, Q3, Q4)*.
- **Structural updates at multiple locations.** This category results in a large amount of disk access as multiple pages in different physical locations are affected. *(Q6, Q7, Q8, Q9, Q7b)* are especially chosen to explore the boundaries of BaseX and MonetDB running on high-end systems. These queries provoke the worst case and will help to identify weak points.
- **Non-structural updates at multiple locations.** The *rename* expression is used to test value updates. This is secondary as value updates are fast to perform *(Q5)*.

Single Location Updates

Q1 . ***insert node (//item)[1] as first into /site***

Q2 . ***insert node (//item)[1] as last into /site***

Q3 . ***delete node (//item)[1]***

Q4 . ***let $i := //item return delete node $i[count($i)]***

Multiple Location Updates

Q5 . ***for $i in //item return rename node $i as "newName"***

Q6 . ***delete node //item/name***

Q7 . ***for $i in //item return insert node <test/> as last into $i***

Q7b . ***let $regionscopy := (copy $s := /site/regions modify (for $i in $s//item return insert node <test/> as last into $i) return $s)***

return replace node /site/regions with $regionscopy Only tested for BaseX as MonetDB lacks support for the *transform* expression. The *regions* element, which is an ancestor of all target nodes, is copied. Hence, all updates are performed in main memory. Ultimately the original *regions* element is replaced with the updated main memory database.

Q8 . ***for $i at $p in //date where $p mod 4 = 0 return delete node $i*** *Date* elements are spread over the document as descendants of *regions-*, *open_ auctions-* and *closed_ auctions-* elements. Using the Tree Map visualization of BaseX helped identifying the *date* element as the most suitable target for distributed structural updates.

Q9 . ***delete node //item/@id***

5.4 Results

	110KB.xml		**1MB.xml**		**11MB.xml**		**111MB.xml**	
Q	BX	MD	BX	MD	BX	MD	BX	MD
1	2	211	1	326	1	312	2	1256
2	1	61	1	72	1	172	2	1152
3	1	48	1	61	1	150	3	1123
4	1	56	1	69	3	476	20	44306
5	1	46	3	59	16	179	181	1490
6	4	65	24	245	1262	2553	114761	68883
7	3	341	37	3315	2410	44528	230706	1643838
8	5	66	46	284	2887	2786	274811	73465
9	4	41	27	49	1218	152	110760	1224
7b	6		42		607		98464	

Table 5.3: Benchmark results, time in milliseconds

5.5 Observations

Single location updates. Although only a single *item* element is deleted, the execution times for Q3 and Q4 vary widely from each other with increasing document size. Due to lazy evaluation, finding the last *item* element takes

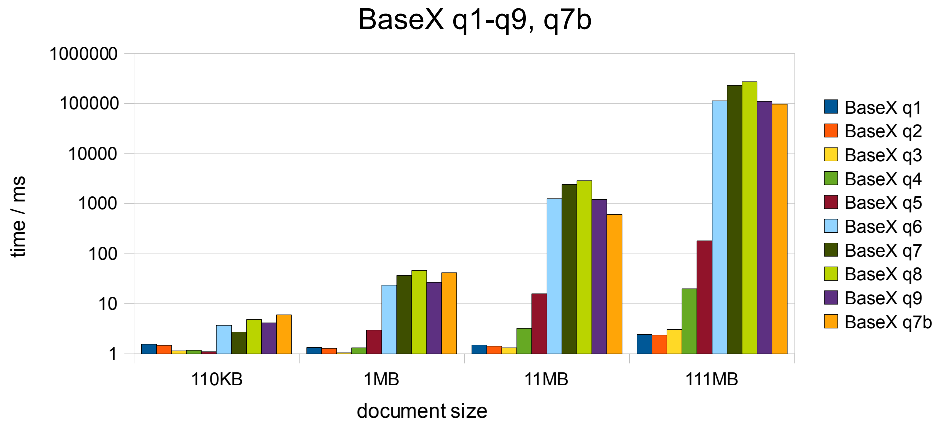


Figure 5.2: test results of BaseX

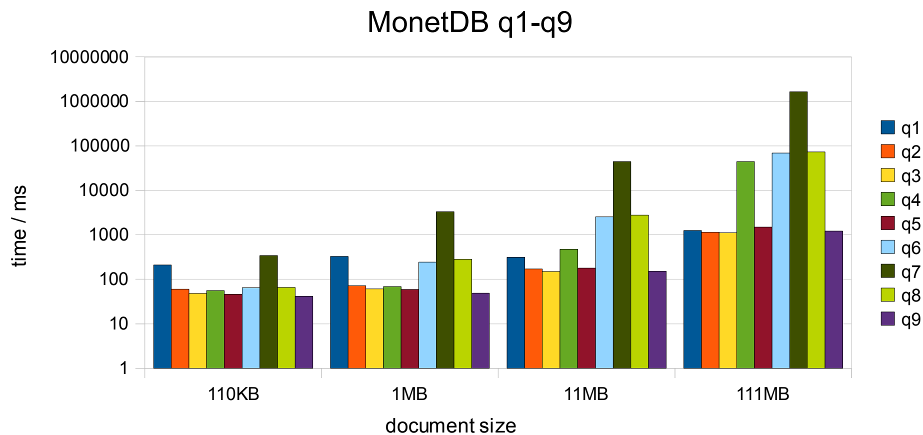


Figure 5.3: test results of MonetDB

considerably more time than finding the first one. Regarding Q3, as soon as the first *item* element is found the evaluation part stops and updates are applied. In contrast, to find the last *item* element during execution of Q4, 21750 nodes are accessed using document 111MB.xml.

There is no measurable difference between the results of Q1 and Q2, as the same target node is accessed. Calculating the exact insert location for Q2 takes constant time using *pre* and *size* values of the *site* element.

Updating multiple locations. Insert operations (Q7) take more time

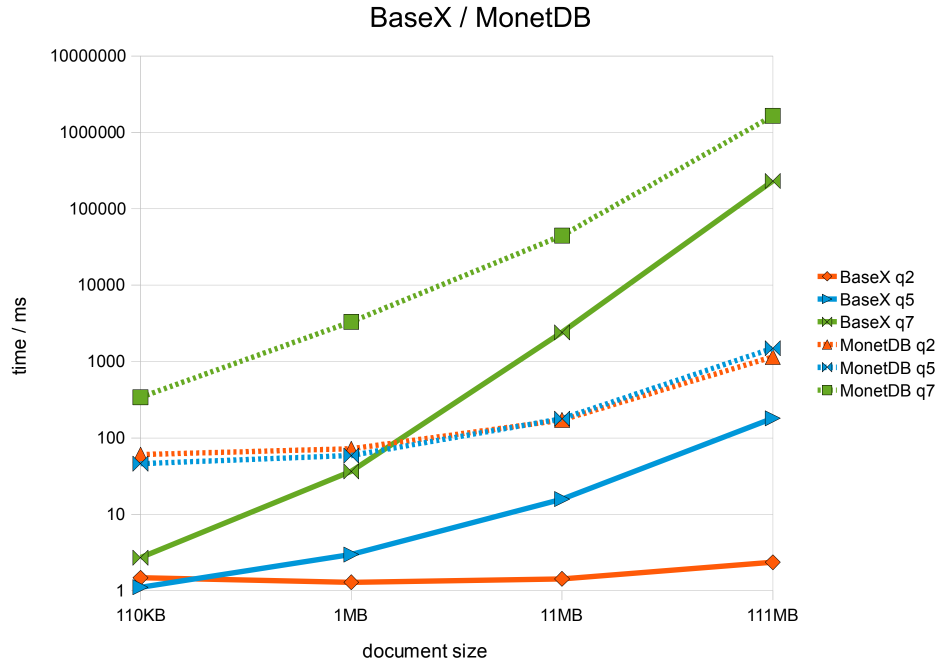


Figure 5.4: comparing results for BaseX / MonetDB

than delete operations (Q6). Deleting a single node effects only a single page. BaseX closes gaps on pages whereas gaps are allowed with MonetDB. This explains the bigger difference between insert and delete for MonetDB. Inserting a single node may effect several pages. As existing tuples are shifted to a new or the following page this results in longer execution times.

Attributes. Regarding Basex, attribute nodes reside on the main table. Operation time is therefore similar between Q6 and Q9. As MonetDB keeps attributes separately execution takes a lot less time.

Value Updates. Updating the value of a node has no effect on the structure of a document (Q5). Adding the new QName to the tag index and changing the tag id for each *item* element takes little time. Therefore overall execution time is short and grows linearly with the number of updated nodes. As, again, more than 20 000 nodes are accessed using document 111MB.xml, execution takes more time than for a single structural update (Q1-Q4).

Scalability. Whereas updating a single location of a document reveals no bigger issues, multiple location updates lead to a super-linear increase of

processing time, see figure 5.4. MonetDB shows a similar behavior for inserts, yet it performs faster on deletes - but at the price of more fragmentation.

BaseX performs very well on single location updates (Q1-Q4) with Q4 forming a sole exception, accessing more nodes to determine the target. Even for larger documents, the answering time is almost unmeasurable and within the range of very few milliseconds.

This once again comes down to the highly optimized architecture of BaseX. We cannot stress enough that BaseX is developed to answer queries in the most efficient way we can think of. We try to ensure this by:

- **Maintaining a minimal set of features.** BaseX does what it is supposed to - and nothing else. This does not mean we sacrifice features for the sake of performance. Our implementation of the XQUF completely meets the requirements including namespaces, XDM consistency and exception handling.
- **Building customized data structures.** All core data structures are built from scratch to serve their individual purpose including lists, hashing structures and trees. This ensures minimized memory usage and maximizes efficiency.
- **Working with primitive data types.** Strings are slow. Byte arrays are used for everything character-involved including query parsing, table representation and more.
- **Rigorously optimizing back end code.** For example bit-shifting and logical operators are used wherever possible.
- **Using no external libraries.**

This way we can ensure that start up time is as short as possible. To answer a query, BaseX parses and compiles the query, opens the corresponding databases, consults indexes to find node ids and finally accesses these nodes on disk via calculating their offset. It must be considered that measurements already include the time for parsing, compilation, evaluation and printing.

As expected, with growing database size and multiple location updates, advantages, regarding execution time, melt away and the overall performance

becomes I/O dominated. Looking at some numbers for the 11MB.xml file explains the super-linear increase of execution time for Q6 to Q9. For the combination of Q6 and 11MB.xml, 2175 *name* elements are deleted. The last */item/name* element in document order has 114 444 as a *pre* value. With 256 tuples on each page, the first 448 pages contain 5 of these elements on average (*2175 / 448 = 4.9*). Currently, updates on a PUL are not aggregated before execution. Each update primitive is therefore applied separately. Consequently the first 448 pages are written to disk 5 times in a row. Aggregating updates for each page could improve performance considerably and is further discussed in Chapter 6.

Increasing page size to 1024 tuples also showed some success. Using the same calculations as above, now 20 */item/name* elements reside on the first 111 pages. For instance, the execution time for Q6 and 111MB.xml decreased by *19%*. With MonetDB, a logical page holds 65536 tuples in default. Together with more advanced caching, this could explain the results of BaseX and MonetDB, lying closer together with growing database size.

A further feature comparison between BaseX and MonetDB reveals some more differences:

- If structural updates are applied with BaseX, the mapping between *id* and *pre* values is lost and indexes become invalid. Resolving this takes some extra time, whereas MonetDB does this on-the-fly. A fast solution is currently developed.
- MonetDB offers a write-ahead-logging that guarantees both, atomicity and durability. Extending BaseX with a similar mechanism is discussed in Chapter 6.
- MonetDB cannot guarantee *consistency* using XQuery Update. For example, duplicate attributes and adjacent text nodes are not detected automatically. The only issues regarding BaseX are *durability* and *atomicity*, which are discussed in Chapter 6.

Although performing admirably, BaseX still offers a lot of potential for optimization and uses very little resources by the way. Looking at table 5.1 might give the impression that XQuery Update produces a high amount of

overhead. In fact, BaseX performs well, even on low-specification machines. As said before, assigning no additional memory and using one core instead of 8 did not reveal any differences.

Updating queries Q1-Q9 are simple and do not generate large intermediate results. Nonetheless Q6 to Q7 are especially chosen to slow down our testing system and, by no means, simulate real life application. Common use cases are no challenge for BaseX and yield excellent results on almost any system. In addition, feedback from the user community attested very fast execution of updates as well.

Main memory approach. As expected, the main memory approach takes less time (Q7b) and reduces page access. Whereas for Q7 all shifts are performed directly on disk, Q7b transfers this expensive operation to the main memory. As BaseX uses little memory in general, using the *transform* expression to process bulk updates is to be considered. On the other side the *regions* element is locked to prevent concurrent access. BaseX currently uses a locking scheme that works on transaction level. Being not an issue at the moment, locking low-level nodes might prevent using advanced concurrency control at a later time. However, execution time of Q7b is expected to be even better after optimizing the *replace* expression which is also discussed in Chapter 6.

Chapter 6

Future Work

6.1 Overview

There are numerous ways to improve performance of updating queries. As the purpose of a database system is to not only handle small amounts of data, but serve multiple clients at a time, features like locking-, restore- and recovery- mechanisms have to be added to the equation. Locking and concurrency is beyond scope of this work. Weiler already provides a glimpse on this in his master-thesis [12].

This chapter gives an overview and proposes some solutions that may help to kick-start future work. Sections cover the following:

1. Ways to support a programmer using XQuery Update by detecting unnecessary operations and applying further optimizations.
2. Revealing weak points and provide solutions based on our test results.
3. Discussion on advanced, yet indispensable features like restore and recovery.

6.2 Optimizations

6.2.1 Fragmentation

Frequent structural updates lead to fragmentation. Gaps on pages after delete operations, or appended pages behind existing ones, as result of an insertion, are forms of fragmentation. Consequently, the answering time of the database system increases, as more pages have to be accessed. In times of low workload, a fragmented database could be copied tuple-wise to resolve this issue.

For example, this can be implemented as a stand-alone operation or be included as part of a backup command. The database itself is not only copied block-wise, but optimized during the process as it is written to a new location on disk. Gaps on pages are filled up and pages reside on permanent storage in document order. To save resources, the database would not have to be copied as a whole. Using the PUL to our advantage, we could keep track of the lowest *pre* value P, that has been affected by updates since the last defragmentation. Optimizing the table is subsequently limited to the logical page that contains P and all following pages.

If heavily fragmented, answering times would improve considerably. Basically, the operation comes down to serializing the current database state and parsing the document subsequently. Processing times for serialization and shredding can be omitted though.

In addition free block bitmaps can help to curb fragmentation in advance. Inserting a subtree it is not always necessary to append a new page behind all other pages. Unused blocks are the result of previous delete operations for example. A free block bitmap keeps track of unused blocks. Thus it attempts to reduce data fragmentation by storing individual blocks of a file in a consecutive manner if possible. This kind of storage management helps delaying the costly defragmentation process.

6.2.2 Optimizing the Pending Update List

With an increase of query complexity, the possibility exists that an update statement contains unnecessary operations. This includes not only operations on node-level, but on document-level as well. Updating and renaming

the same node during a snapshot ends with the node being deleted (see 'order of updates', figure 3.2). While this example can be detected easily, the hierarchical nature of our data complicates the issue. If a node n is deleted or replaced, all update primitives, that have a descendent of n as a target, are without effect. Depending on document size and complexity, updating transactions can be expensive. Identifying superfluous operations is therefore of importance.

Thanks to the customized table encoding of BaseX, this can be taken care of without adding any overhead at all. Before updates are applied, the PUL aggregates all pending node state changes. We now apply the following steps:

1. **Sort** the PUL ascending, depending on the *pre* values of the target nodes.
2. Start with the first *pre* value P.
3. **If** P is not target of a delete or replace, skip to the next value.
4. **Otherwise** determine the size S of P. S is the size of P and its subtree.
5. **Delete** gradually each following *pre* value on the sorted PUL, that falls within the range of P and $P + S - 1$, as these nodes are descendants of P.
6. **Continue** with step 3 and the next value or stop, if the last value is reached.

Using the *size* column of the table, *pre* values within a subtree of any node are detected easily. Let n be the size of the PUL. Applying this algorithm takes $\mathcal{O}(log\ n + n)$ time, including sorting and linear traversal of the list. In addition, the amount of page access correlates with the number of nodes that are replaced or deleted.

6.2.3 Block-wise replace

Currently the replace operation is a compound of delete and insert. That means pages are accessed twice during a replace. Query 7b for example could profit a great deal if I/O is reduced. One solution is to overwrite existing

nodes explicitly. Up to now, replacing a node together with its subtree is slower than it should be. First, nodes are deleted and possible gaps on the page are filled by shifting following tuples. After this, inserting the new tree results in an additional shift of following tuples.

Yet, the concept of pages to reduce shiftings on disk can be easily exploited. If the number of replaced nodes is equal to the number of inserted nodes, tuples could be replaced sequentially. If the two trees are of different size, the following has to be done:

1. **Locate** the page and position p of the node to be replaced.
2. **Let** t be the smaller tree (insert, delete) and s its size.
3. From position p to p+s-*1* **overwrite** the original tuples with the new ones in document order.
4. **If** t is the tree which is replaced (hence the smaller one), **insert** the remaining nodes of the insertion tree at p+s. Tuples might have to be shifted and/or one or more new pages appended.
5. **Else** if t is the insertion tree, **delete** the remaining nodes of the original tree. Tuples might have to be shifted and/or one or more pages deleted.
6. Update the *size* value of **ancestor** nodes.

Using main memory for expensive updates could further profit from this solution. This has already been tested with Query 7b (see Chapter 5).

6.2.4 Page-wise updates

Taking a closer look at the test results revealed that mostly unnecessary page access is responsible for super-linear scalability. This is especially true for updates changing the structure of a document in multiple locations. Block-wise replacing is a way to reduce I/O for certain situations. A similar approach that builds upon this helps optimizing all kinds of structural updates. The basic idea is to summarize operations for each logical page and execute them as a unit. Structural updates would not cause shiftings on disk any longer, but are performed in main memory. A first step is to divide the PUL into

sections with each section containing the pending changes for a logical page. Yet, we are facing several complications:

1. Affected update locations For convenience we take the largest subtree of a document to explain this issue. The root node together with its descendants occupies all pages of a table. If a node is added to the root via an 'insert into' statement, an update primitive is created which target node has *1* as a *pre* value. Obviously, this *pre* value is not necessarily the location which the actual update is applied to. New nodes are added behind existing ones.

Up to now, the PUL helped organizing node state changes using the *pre* value of targets as identifier. As shown before, this simplified to meet specification requirements regarding XDM constraints. The elimination of unnecessary operations (tree-aware updates) is also based on this. Unfortunately, the page-wise aggregation of updates requires knowledge about the actual locations where structural changes are applied. We therefore have to introduce a mapping between target *pre* values and *affected pre* values. With *pre* as a target the following table shows the actually affected position for each expression:

operation	**affected position**
rename	*pre*
insert before / after	*pre-1* / *pre+size(pre)*
insert into as first / last	*pre+1* / *pre+size(pre)*
replace	*pre*
delete	*pre*

This knowledge can be used to sort the PUL with regards to affected positions. Updates can then be aggregated for each page.

2. Updates on non-existing subtrees Updating the subtrees of deleted or replaced nodes is another issue. As we apply updates bottom-to-top, target nodes on the preceding axis of our current target are updated at a later point. Allowing this increases page access. Deleting superfluous primitives has been explained before, see the section about tree-aware updates.

3. Subtrees spanning several pages Applying the algorithm for tree-aware updates makes sure that unnecessary operations are removed from

the PUL. Still, there is the possibility that a node is deleted which spans more than one page. Take an element node for instance together with its *pre* value P. S is the size of P. Deleting P requires the following steps:

1. Determine the logical page of P and its position L on this page.
2. From L onwards delete all entries from the current page and write it back to disk.
3. Delete the following pages that are completely occupied by nodes that are descendants of P.
4. On the last page which contains part of the subtree of P, delete these nodes and shift following nodes to the beginning of the page. Write page back to disk.

After clearing things up, applying page-aware updates becomes simple:

1. Apply **tree-aware-updates** algorithm.
2. Calculate the **affected** *pre* value for each update primitive.
3. Determine the corresponding updates for each logical page and create a list L for each target page. For example, a page holds *256* tuples. If the *affected pre* value of a primitive *UP* is *260*, *UP* is added to L of the second page.
4. Sort all L descending.
5. Read the page which corresponds with the first L.
6. Apply updates on this page in main memory bottom-to-top.
7. Write page back to disk.
8. Continue with the next L.

What exactly is the benefit of this strategy? Using the PUL, the calculation of affected *pre* values requires linear time with regards to the number of update primitives. So far, if a logical page is target of multiple updates, it is written to disk individually for each atomic change. That basically wastes a

lot of IO. Some calculations on this are given in Chapter 5. Another benefit of page-wise updates is the simpler detection of adjacent text nodes, see Chapter 4. As pages are touched only once, this can be taken care of in memory without adding additional IO like the current solution.

However, only future tests will reveal the actual benefit of this approach. This is due to caching mechanisms on hardware- and operating-system level, together with BaseX being developed in Java.

6.3 Rollbacks and Recovery

Things can go wrong at every time. Thus rollbacks and recovery mechanisms are important for a database system. The ACID principles are a guideline for this. BaseX currently provides a basic restore mechanism, that enables the user to restore backed up databases. Providing a certain security, more or less data gets lost if a database is rendered unreadable and not backed up regularly. Depending on the size, saving a database state can take a significant amount of time and is not suited for regular use.

First we have to define what types of errors are likely to occur and which are of interest for us. The main goals, regarding development of BaseX, are simplicity and efficiency. As today's systems get more and more complicated and diverse terminology leads to increasing confusion, I deem necessary returning to more basic work for reference. Some time ago, Haerder and Reuter made an effort to clear things up by providing a terminological framework describing recovery schemes for transactional database systems [9]. In principle they divide database failure into three categories:

1. **Transaction Failure** A transaction has to be set back, because it is not committed regularly. For example a deadlock can lead to one or more transactions that need to be rolled back.

2. **System Failure** is caused by code errors or hardware failures. 'Someone pulling the plug' falls into this category. Contents of main memory are lost in effect.

3. **Media Failure** describes fatal damage (i.e. head crashes) and damage caused by operating systems or controllers to secondary storage. Sheer redundancy helps to prevent data loss.

BaseX guarantees *isolation* using a locking scheme that allows parallel access for read-only transactions. If a transaction changes the state of a document, all other transactions are put on hold. This eliminates the risk of deadlocks and dirty reads completely.

Every snapshot is treated as a separate transaction. By definition, a database is in a *consistent* state before and after the successful execution of a snapshot [4].

So the only principles endangered by database failure are *atomicity* and *durability*.

Rollbacks make it possible to undo transactions that have not been completed regularly. The current locking scheme eliminates situations that require a rollback. Hence this feature does not contribute to ACID conformity. But it could be misused to help a user abort expensive transactions. Preparing page-aware updates leaves us with a sorted PUL regarding the affected positions of a document. If a transaction is aborted, the PUL helps to revert changes that have already been applied. Effects of update primitives are reversible. To stop a transaction, complements of already applied node state changes on the PUL have to be carried out in a top-to-bottom manner. This way we can keep *pre* values as identifiers. An undo presumably takes about the same amount of time as the initial execution. It is only beneficial if the user consciously wants to abort expensive transactions.

Recovery on the other hand is substantial. Using BaseX, data can only be lost in case of a system or media failure. Common solutions that are used in other systems include a base version of a database together with a log file, which stores successfully finished transactions in a sequential manner. Facing a failure, the database is restarted and a working copy of the original database created. Re-applying logged transactions then leads the database back to its most recent state before crashing.

BaseX already creates a log file if the server/client architecture is used. It is not sufficient to write a transaction together with its query statement to the log file. To prevent dirty reads for example, source information has to be stored as well. Therefore a log file not only consists of the statements. XDM instances of copied nodes and new values need to be stored as well. Furthermore XQuery Update enables the user to access multiple documents within a single query. As a result, multiple databases can be destroyed by

the same transaction. A log file must consequently be kept for each database individually. Another thing worth considering is to store log files in a different physical location. Otherwise media failure can be fatal.

Adding a recovery component can turn out to be expensive. Each primitive together with its source data is to be serialized in advance. Worst case, this means that complete documents are written to disk before anything else happens. Moving this operation to the end may save some time. Recovery itself can also take a considerable amount of time, depending on scale and complexity of the application. A solution that performs faster on the recovery part is probably in conflict with the highly efficient nature of BaseX. Pagewise versioning for example helps faster backups, but is more difficult to realize.

Chapter 7

Conclusion

Goal of this thesis was to extend a cutting-edge native XML database with the widely accepted updating standard XQuery Update. It was possible to add the new module in a way, that it does not negatively interfere with the existing system. Conforming to the standard, the implementation is fully applicable in a productive environment and yields impressive test results.

To reach this goal, the basics of a native XML database system were explored. With a sequential encoding scheme, based on a pre-order traversal of the document tree as the foundation, further analyzation of the XQUF revealed several points of interest. The special table encoding of BaseX leaves room for optimization. Structural document changes can be carried out in a bottom-to-top manner, which renders updating of the *pre-id* mapping within a transaction unnecessary. On the other hand, some features of the standard complicated the implementation process. Simple and efficient solutions were provided for text node adjacency and usage of the *'insert before'* statement. The proposed architecture is compact and efficient and therefore exhibits the same qualities as the rest of the system.

Benchmarking attested excellent results for real-life application. Queries, especially designed to explore weak points of the current solution, revealed super-linear scalability for multi-location structural changes.

Whereas the query part works close to the limits, manipulation of data takes more time than it could. Performing each atomic update individually, logical pages on disk are accessed more often than necessary. Page-wise aggregation

of updates together with tree-aware updates could therefore improve performance by magnitude. An algorithm is proposed, that exploits the sequential table encoding once more to its full advantage. Other optimizations include a free block bitmap, that optimizes the allocation of new logical pages and a solution to curb fragmentation of the database.

Fulfilling the ACID properties is vital for a database system. With the integration of a basic and efficient recovery mechanism in the future, BaseX fully conforms to this principle and is well equipped to serve data-critical and time-critical applications. Whereas a basic solution can be added easily, a more sophisticated approach not only requires more time. It has to be decided, whether BaseX wants to follow this route. Simple solutions may suffice, as BaseX has not been designed for highly concurrent environments. The same applies for an advanced way to handle concurrency.

Bibliography

[1] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, et al. XQuery and XPath Full Text 1.0. W3C Candidate Recommendation. http://www.w3.org/TR/xpath-full-text-10, July 2009.

[2] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Monetdb/xquery: A fast xquery processor powered by a relational engine. In *SIGMOD*, 2006.

[3] P. A. Boncz, S. Manegold, and J. Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery. In *XIME-P*, 2005.

[4] D. Chamberlin, M. Dyck, D. Florescu, J. Melton, J. Robie, and J. Siméon. Xquery update facility 1.0. http://www.w3.org/TR/xquery-update-10, Jun 2009.

[5] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. Xquery 1.0 and xpath 2.0 data model (xdm). http://www.w3.org/TR/xpath-datamodel, Jan 2007.

[6] C. Grün. *Storing and Querying Large XML Instances.* PhD thesis, University of Konstanz, Konstanz, Sep 2010.

[7] C. Grün, S. Gath, A. Holupirek, and M. H. Scholl. Xquery full text implementation in basex. In *XSym*, pages 114–128, 2009.

[8] T. Grust. Accelerating xpath location steps. In *SIGMOD*, pages 109–120, 2002.

[9] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. In *Computing Surveys, Vol.15,No.4*, 1983.

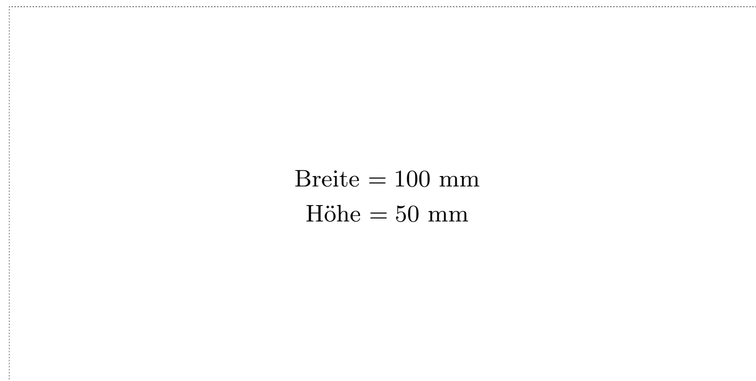
[10] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: Insert-friendly xml node labels. In *SIGMOD*, pages 903–908, 2004.

[11] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Updating the Pre/Post Plane in MonetDB/XQuery. In *VLDB*, 2002.

[12] A. Weiler. Client-/server-architektur in xml datenbanken. Master's thesis, University of Konstanz, Konstanz, 2010.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —