Treetank, Designing A Versioned XML Storage

Sebastian Graf, University of Konstanz

Marc Kramis, Seabix AG

Marcel Waldvogel, University of Konstanz

Abstract

XML underlies the same constant modification scenarios like any other resource especially in flexible environments like the *WWW*. Therefor intelligent handlings of versioned XML are mandatory. Due to the structural nature of XML, the efficient storage of changes in the data and therefor in the tree needs new paradigms regarding efficient storage and effective retrieval operations. We present a node granular XML versioning approach which relies on the independence of the storage and the versioning system. Different layers which have the ability to satisfy specific aspects of a node-granular versioning storage guarantee this independence. Results prove that our architecture offers efficient handling of consecutive changes within all modification scenarios while not restricting XML regarding its usage. Hence, our prototype system handles even huge XML instances while ensuring equal access to each revision of the data.

Introduction

XML data as a text based format underlies heavy modifications especially due to its designation as *ligua franca* in the world of *WWW*. Its flexible usages in configuration files, representation formats and direct accessible data resources suffers from consecutive insert, remove and changing operations on the data and therefor on the tree structure.

Even if XML is easy modificable e.g. with the help of XQuery/Update expressions, only few approaches exist to handle different versions of the same data with focus on the storage. Current versioning approaches are surely able to provide efficient methods to handle any kind of changes but regarding XML with its tree- and text representation, there are only a small systems available to work with different versions of the same data in a scalable way which is necessary especially for large XML databases.

We therefor present an architecture for an integrative versioned XML database called *Treetank*. Based on the different workloads and the flexible usages of XML, we motivate an independence between the representation of the tree and its storage. This independence offers us high adaptability and constant access to all versions with minimal overhead regarding storage and access time.

Related Work

Versioning left the area of being a tool for maintaining source code like CVS[cvs]. Even if CVS and modern successors like SVN[svn] are able to fullfil versioning of XML based on their text-representation, they do not make any use of the structure.

XML-Versioning nevertheless became common within the last years. Zholudev and Kohlhase present with TNTBase[tnt09] an integrative XML versioning system. This system takes SVN[svn] as a base and offers an integration of the versioning functionality with the Berkeley DB XML[bdbxml] database. The overall XML structure is thereby maintained by two established and well maintained software projects even if the versioning itself takes not place on node level.

Another approach regarding XML version is the *Time Machine for XML*[tm10]. *XQuery* PULs represent deltas between XML versions. A specialized data-structure called π–tree cares about the storage of the versioned nodes. Since this approach relies on an ORDPATH-based [ord04] encoding,

the underlaying page architecture offers clustering to overcome the linear search for the suitable π-nodes and revision. Therefor, UBCC [ubcc00] is introduced to minimize this effect. UBCC works with thresholds regarding usefulness. If a defined threshold is reached, the pages are rearranged regarding their contents. This behavior can result in peaks and synchronization issues related to the read- /write-performance, so we choose a different approach which makes use of the append-only paradigm as described later.

ZFS[zfs03] represents a file system with features which we consider valuable for our native, versioned XML storage. ZFS enables easy handling of versions, called *Snapshots*, and has its focus on both, the scalability regarding large datasets and the integrity for different versions of the same data. Hence, its page architecture acts as a starting point regarding design decisions in our system as explained later.

Motivation of three core concepts of a versioned XML storage

The motivation for a node granular versioning system results in features for an efficient node storage and retrieval architecture. Hence we want to motivate three core aspects which act as a base for our proposed system.

1. The nodes must contain all information about their content and their position in the overall structure.

   Regarding huge XML databases, the kind of storage of the XML must respect the tree structure. Several approaches encode the position and the content of a node related to their overall position in the tree to support efficient handling of the inherent data. Regarding versioning of nodes, such an encoding must be specialized in a way that regardless of the kind of modification a minimal number of nodes are touched to represent the change in the tree due to scalability reasons. An adaption of the entire tree structure represented by one delta would not be practicable within a node-granular versioning system.

2. The position of a node in a tree must be flexible regarding its position in the storage.

   Even if the position of a node in a tree is fixed over time, we motivate an independence of its role and its mapping to the storage. Modern storage systems use a page-based storage system for organizing content on disk. Nevertheless, such page-based storage layers must be adapted to support versioning in a native way.

3. Changes to the stored nodes must rely on a convenient and confident transaction system.

   Based on the encoding and the storage, changes related to the tree structure must rely on a defined transaction system. Due to the atomicity of versions related to consecutive modification requests, we believe that a transaction layer with a distinction between read- and write-accesses is the base for a clean and trustable versioning system.

These three motivations build the pillars for our XML versioning system. Each pillar is covered by one separate layer. The node-layer covers our first motivation about the node encoding. All aspects regarding the content and the representation of a node represent this layer. The page-layer satisfies our second motivation. This pillar of our system covers the kind of storage of nodes plus the arrangement of pages related to each other to version the data. The third motivation results in our transaction-layer. This layer is responsible for the accesses and the interfaces for implementing services. The next section describe these three layers in detail since they represent the core aspects of our architecture called *Treetank*.

Architecture of Treetank

The architecture of *Treetank* consists out of three main modules, called layers. Even if these layers are arranged in a horizontal manner, they can be combined with different peculiarities to ensure independency and flexibility. The node-layer, which represents the XML nodes, cares about the

encoding and the node information while referencing and dereferencing of nodes from and to the persistent storage is part of the page-layer. The transaction-layer acts as the bridge between both and offers access to the data to external resources, interfaces and applications. We will further describe these three layers and how they interact with each other in detail:

Node-Layer

The first core layer of *Treetank* is the node-layer. Since we motivate node-granular versioning, we ensure a suitable node representation within our architecture. For flexibility and scalability reasons, *Treetank* maps the different node kinds of XML regarding their specific characteristics.

Figure 1. Different supported node-types

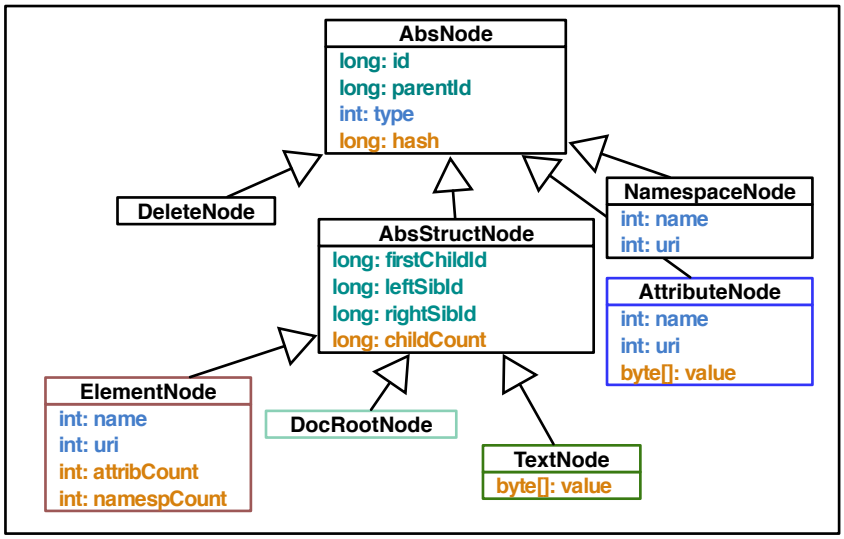


Figure 1 defines all supported node-types in *Treetank*. As denoted, there is a distinction between nodes which represent content only and nodes which have additional structural properties.

All nodes in our system have a fixed reference denoted by an unique identifier. This id, as denoted in the *AbsNode* in Figure 1, is valid for one node in all versions. It satisfies the following three requirements:

1. The unique identifier acts as a direct reference for other nodes. As described later, our structure relies on local relationships only which are built upon these unique identifiers.

2. The unique identifier is the pointer for the nodes in our page-layer. Based on this identifier, our system retrieves and stores any node.

3. The unique identifier offers direct access to specific nodes without additional encoding. Even if the identifier is not order aware like e.g. ORDPATH [ord04], our identifier ensures direct access to nodes for external resources.

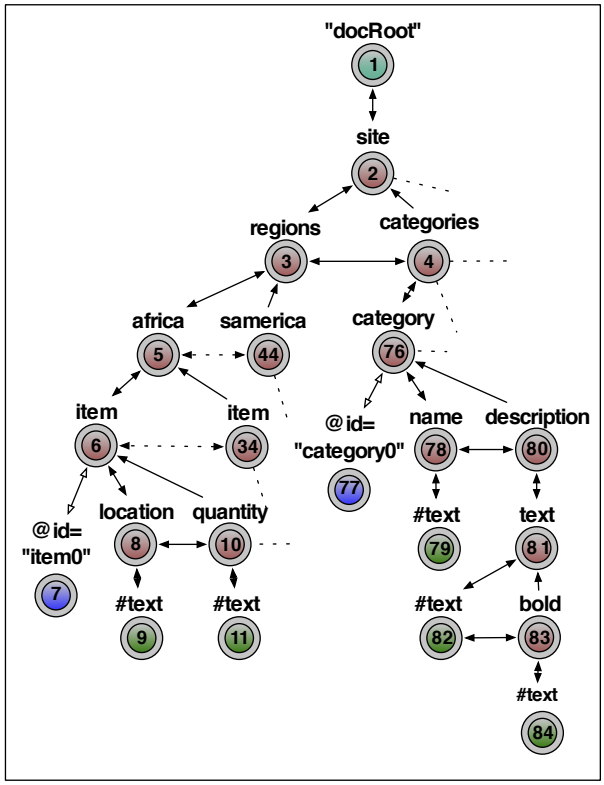
Besides the unique identifier, all nodes holds a reference to their parents. Additional, structural nodes like *ElementNodes* and *TextNodes* hold furthermore a reference to their left-siblings, to their right-siblings and to their first-childs. All of these fields are denoted as *structural attributes* and colored cyan in Figure 1. These pointers build the backbone for our structure as explained later.

Regarding the concrete content of a node, each node holds furthermore a specific type key. The type maps the XSD type to a concrete node. Since we rely on a central mapping of common tag-names, uris and types, the field stored within each node is only a pointer. The target of this pointer is part of the page-layer as described later. Additional, *name* and *uri* of *NamespaceNodes*, *AttributeNodes* and *ElementNodes* are pointers to this central string store as well. The pale blue color of the attributes on Figure 1 denote these *referential attributes*.

Contents which cannot be referenced, are stored directly. One example is the hash value. The tree structure generates the hash with the help of the recursive relationship between nodes. Therefor each hash value of a concrete node guards the integrity of its corresponding subtree. This offers us the possibility to track changes in our structure without expensive diff-computations on subtrees. Other attributes representing direct data are the count of associated nodes like children, attributes and namespaces in *ElementNodes* as well as the concrete text denoted as value in *TextNodes* and *AttributeNodes*. These fields are called *content attributes* and colored orange in Figure 1.

As already mentioned, the structural backbone of any XML in our architecture is a specialized node encoding. This encoding consists out of pointers to the unique identifiers to the left-sibling, right-sibling, first-child and parent. Figure 2 shows such an encoded XMark[xma02] instance: Since *ElementNodes* and *TextNodes* are structural, they embody the structure of the tree due to their stored relationship to their specific neighborhood, e.g. each "category" node holds a reference to its parent ("categories"). Additionally, the first-child ("name") is referenced as well as the related left- and right-sibling nodes ("category"), if existing. Attributes like"id" are not treated as structural nodes due to their special association to *ElementNodes*. The encoding uses specialized pointer for those associated nodes. The nodes in Figure 2 map regarding their colors to the different node types explained before.

Figure 2. Example Encoding of XMark-Instance in Treetank



Our encoding offers constant scaling regarding adaption of the structure within node insertions. Expect the insertion of the new child-node "namerica" as a child of the existing "regions" node in Figure 2. This node should be inserted as a left-sibling node of "samerica". This results in the following adaptions:

- The child-counter of the parent is increased.
- The hash-value of the ancestors are adapted.
- The right-sibling pointer of the new "namerica" node is redirected to the existing "samerica" node.
- The left-sibling pointer of the new "namerica" node is redirected to the existing "africa" node.
- The right-sibling pointer of the existing "africa" node is redirected to the new "namerica" node.

- The left-sibling pointer of the existing "samerica" node is redirected to the new "namerica" node.

The adaptions on the encoding take only place on related nodes. Even if we insert huger subtrees under the new node, the described adaptions remain the same. Regarding the removal of existing nodes, the locality of adapted nodes is ensured as well. If we want to remove e.g. the first "category" node including its subtree, we face the following adaptions:

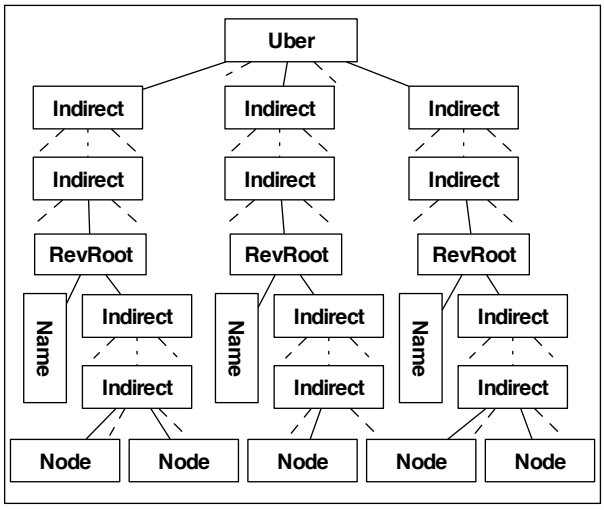
- The child-counter of the parent is decreased.
- The hash-value of the ancestors are adapted.
- The first-child pointer of the parent is redirect to the existing "category" node which acted as a right-sibling of the node to be removed.
- The left-sibling pointer of the existing "category" node which acted as a right-sibling of the node to be removed is deleted.

These examples show that insertion and removal operations generate only minimal impact to the overall structure. Due to the locality of our approach no global adaption takes place. Hence we consider the pointers necessary for an integrative node versioning system. Furthermore the locality of our approach offers us great possibilities to provide iterators based on the pointers and therefor on the structure, e.g. providing XPath-Axis is straight-forward based on the pointers between the nodes. Combined with the unique identifier, exploration of even huge structures takes place based on known and cached nodes denoted by their identifier or on the pointers of the nodes. The described node-layer offers great extensibility for new nodes (e.g. *CommentNodes* which are not supported yet) as well. Based on the modular structure, we further have the ability to equip our existing architecture with permissions and encryption features on nodes directly. Our node-layer therefor not only allows us to version with fine granularity but also to extend and adapt our architecture in a flexible way.

Page-Layer

The second layer which satisfies our motivation of flexibility for versioning is the page-layer. Since the node-layer itself has no functionality to version any data, the storage takes care about the handling of different versions. Yet, the versioning functionality is not easy adaptable to common sequential paging architectures where ensuring constant access and minimal rearranging of data with respect to consecutive changes is mandatory. Therefor, we introduce a more complex structure which offers same access to all data with minimized redundancy and append-only architecture. This results in a tree structure of pages motivated by the page layout of ZFS[zfs03]. The following Figure 3 shows the concrete architecture:

Figure 3. Page-Layer Architecture



These page kinds have different attributes and intentions:

- The *UberPage* is the main entry point for each store and retrieval process. Therefor it is the only page which stays synchronized at all times for ensuring integrity for all versions. It holds the global version counter as well as direct references to *IndirectPages* and hence indirect references to *RevisionRootPages* and *NodePages*.

- The *IndirectPage* acts as a container for pointers to other pages. *IndirectPages* occur between the *UberPage* and *RevisionRootPages* as well as between one *RevisionRootPage* and its specific *NodePages*. Both times, *IndirectPages* are aligned in a layer of 5 pages. Based on a pointer-set of 128 in each *IndirectPage*, we are handle to support 128^5 revisions and nodes at the moment.

- The *RevisionRootPage* represents one concrete version of the data. Since each *RevisionRootPage* acts as a root for the data modified or created in this concrete version, it provides one slide regarding the overall dataset. Each transaction holds a fixed set of *RevisionRootPages* to access the data.

- The *NodePage* contains all concrete nodes. Therefor, *NodePages* are always leafs in the page-layer. Starting from a *RevisionRootPage*, the unique identifier of a node tracks the related *NodePage*. *NodePages* symbolize the append-only principle. Hence they do not offer possibilities to rearrange nodes between each other.

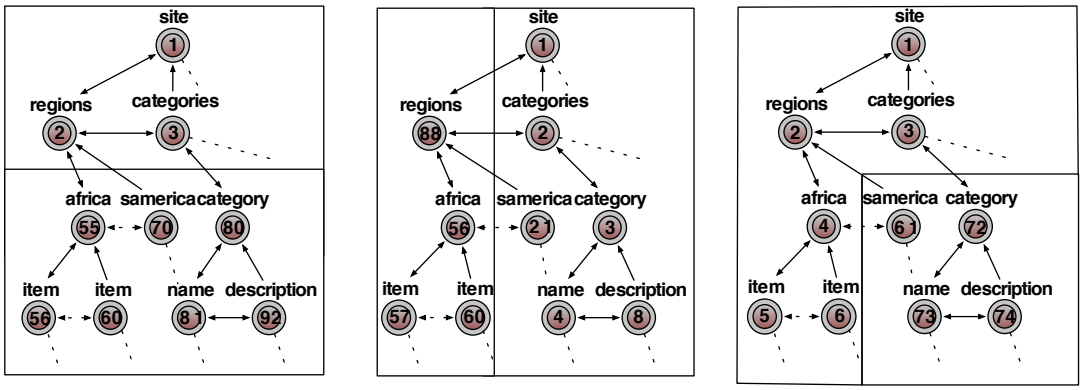
- The *NamePage* is the central storage of common tag-names, uris and types. Regarding its architecture it acts as a flexible sized hashmap. Since tag-names mainly consists out of a fixed set related to common XML instances, we decrease the overhead of storing atomic strings. Instead, we reference the strings in the *NamePage* related to the corresponding revision and point to them over unique keys in the related nodes.

The proposed architecture works with an append-only mechanism. The *NodePages* are filled differently related to the used versioning approach. That means that with differential versioning, the *NodePages* in the first revision store all nodes. Afterwards, only the adapted nodes are stored in the *NodePages* associated to the first *RevisionRootPage*. Regarding incremental versioning, all changes are stored in the *NodePage* associated to the last *RevisionRootPage*. This results in a suitable set of *RevisionRootPages* related to each transaction. Even if we work with an local encoding to minimize the impact of deltas to our structure, we further adapt our system to fit perfectly write- or read-performance demanding workloads.

Since the kind of versioning reflects the load of the *NodePages* without any knowledge about the overall tree structure, every transaction must hold the suitable set of *RevisionRootPages* to reconstruct the structure. This set depends on the current versioning algorithm. For incremental versioning, the set of *RevisionRootPages* consists out of all pages starting with the last full-dump. Regarding differential versioning, only the last version and the last full-dump is selected. Starting from the set of *RevisionRootPages*, each node is referenced in the same manner over the layer of *IndirectPage*s. This results in the same performance regarding all versions and all nodes. *DeleteNodes* are inserted within removal operations to ensure the correct reconstruction of the tree distributed over multiple *RevisionRootPages* and their subtrees.

The aspect of independent dereferencing of nodes regardless to its versioning and storage is also founded on the unique identifier of the node. This identifier acts not only as the base for building up the structure, it also represents the reference to its storage. With the help of the concrete node identifier, the related *NodePage* is identified. Since an identifier is valid for only one node regarding the entire lifespan of the database, this referencing architecture fits perfectly to our append-only approach. If one *NodePage* reaches its maximum number of nodes, upcoming nodes are stored to the next *NodePage*.

Figure 4. Different NodePages with similar content



This results in different distributions of nodes along *NodePages* based on their order of insertion. Figure 4 shows the same tree stored on different pages since the structure is inserted into our system within different orders. However, based on the unique identifier of each node, all pages are dereferenced in the same manner and in the same time. Since forecasting of all upcoming modifications between revisions is impossible, we believe that the flexibility and adaptability between encoding and storage is an elegant yet effective way to tackle this problem. With our approach of storing append-only, we are further not facing common problems like fragmentation or rearranging of the pages.

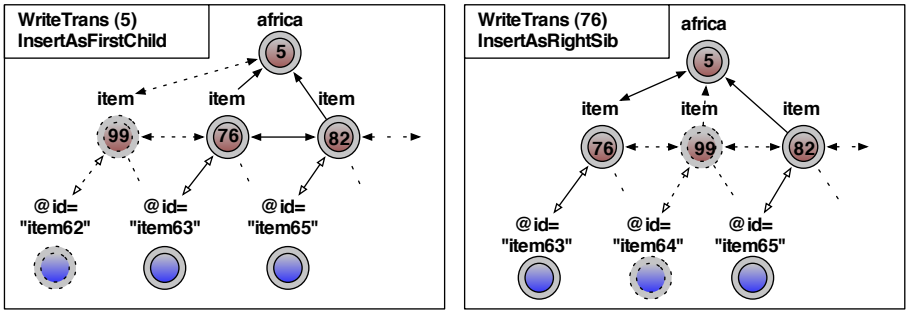
Even if we generating constant overhead regarding referencing and dereferencing of nodes compared with existing paging techniques, our approach simplifies most of the issues that come along with a versioning system. Together with the optimized node encoding, we reduce common drawbacks like fragmentation and variable access times. The introduction of the unique node identifier in the node-layer enables us furthermore to dereference any node within constant time over a fixed number of *IndirectPages*.

Transaction-Layer

The third pillar of our system is the transaction-layer. This layer acts as the bridge between node-layer and page-layer since it covers the access to the nodes regarding read- and write-operations. Since modifications in the tree regarding content and structure result in an adaption of both, nodes and pages, we reduce the number of write accesses to one at one time. This results in a distinction between *ReadTransactions* and *WriteTransactions*.

ReadTransactions are the common way to work with *Treetank* when it comes to query and retrieval operations. Each *ReadTransaction* is bound to a version or timestamp to get exact the requested data. Based on the node encoding, a *ReadTransaction* is able to navigate in the tree following the pointers from one node to the other. This results in simple cursor operations where the transaction iterates either on the pointers or jumps random-access like to already known nodes denoted by their identifier. More concise, each *ReadTransaction* has the ability to move from one node to its left-sibling (if existing), to its right-sibling (if existing), to its first-child (if existing) or to its parent. These operations differ from the concrete node-type the cursor is bound at the moment of its operation e.g. the move to a left-sibling is not possible if the cursor points to an *AttributeNode*. All *ReadTransactions* are not only independent regarding their version but also regarding threads. Each access to the data is thread-safe which results in a highly scalable system with respect of the overlaying framework.

Figure 5. Insertion of nodes as first-child and as right-sibling



In opposite to the *ReadTransactions*, *WriteTransactions* encapsulate modifications on the content. These modifications cover both, content related updates as well as structure related operations. Each *WriteTransaction* has the same operations than common *ReadTransactions*: They navigate the tree based on normal cursor-move operations. In addition they offer methods to insert new elements. Withal, the insertion of structural elements takes place either as insertion as a first child or insertion as a right sibling. Insertion of nodes result commonly in:

- the creation of a new node. This new node gets its unique identifier while creation and is appended at the end of a *NodePage* using the append-only paradigm.

- the adaption of the neighborhood. The related nodes are not recreated. Instead, only the pointers are adapted. Nevertheless, a fresh copy of the nodes are stored within a *NodePage* related to the current *RevisionRootPage*.

- the temporal persistent storage of the touched *NodePages*. These pages are cached in a persistent transaction-log. This transaction-log is flushed to a new revision when all modifications are finished denoted by a *commit* command. Triggered by this command, a new subtree is created in the page-architecture to make the new revision visible to other transactions.

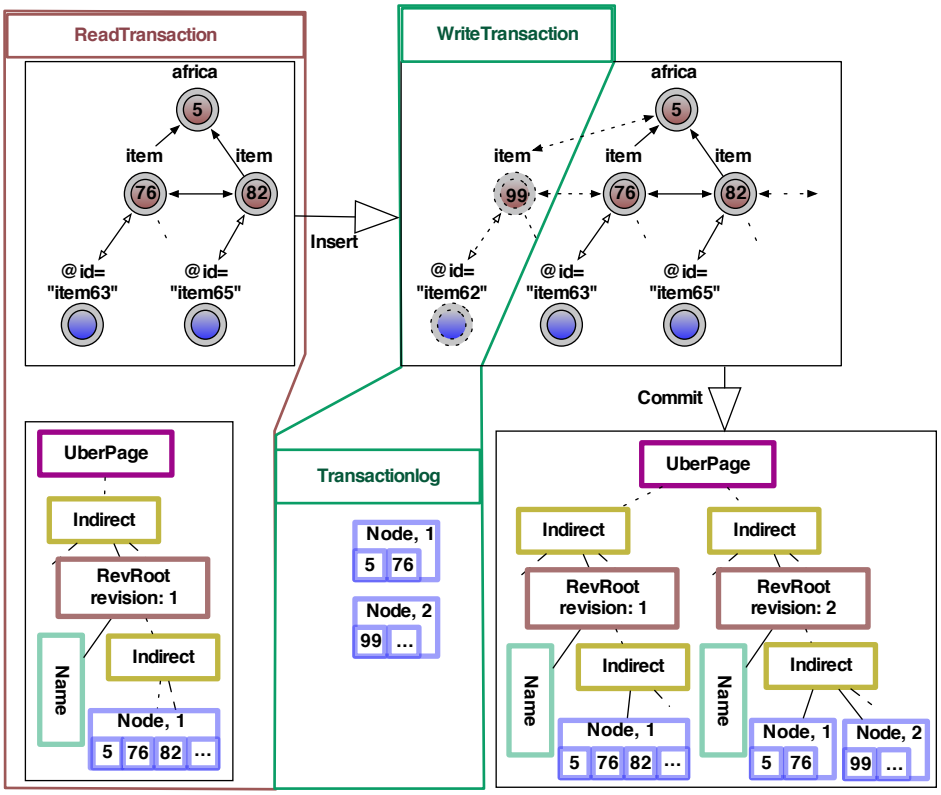
Figure 5 above shows an example of such an insert operation in the tree. The *WriteTransaction* is moved to the node where an insertion is wanted. In the left picture, the cursor points to the node with the id "5" and a new subtree is inserted as first-child. This operation adapts all of the dotted pointers. As clearly visible, this results only in a constant set of nodes touched within this operation. In the right picture, the cursor points to the node with the id "76" and a new subtree is inserted as right-sibling. Again, the dotted lines show the pointers to be adapted.

The cursor based transaction-layer offers direct access to any nodes as well as insertion of new content anywhere in the tree. Due to the independence of the cursor architecture based on our node encoding, the transactions are encapsulated from any versioning functionality (except regarding its instantiation). Furthermore it provides an extensible and easy interface to be used by any kind of query- and modification interfaces.

Interworking between Node-, Page- and Transaction-layer

Together these three layers represent the main functionality of our system. An example of the interaction of all layers is shown in Figure 6

Figure 6. Interaction of the three layer within insertion operation



Already existing data in revision "1" is retrieved with the help of a *ReadTransaction*. The red area denotes this transaction which is bound to the pages under the *RevisionRootPage* with revision "1". If an insertion of a new subtree occurs, the *ReadTransaction* must be replaced with a *WriteTransaction*. The green area denotes this running transaction. Within this transaction, a new subtree with an "item" element extends the existing tree. This extension results in one new *NodePage* with the new node and an adaption of the existing *NodePage* with the nodes which are modified due to their updated pointers to the new "item" element. Both pages are stored in the transaction-log while the *WriteTransaction* is running. Within the commit-command, the changes in the page-layer from the transaction-log become valid by inserting both, the modified and the new *NodePages*, under a new *RevisionRootPage* which represents the new revision "2".

This example shows that our transaction-layer combines the independent node- and page-layer regarding retrieval, storage and committing of data since it relies on both, the node encoding on the one hand and the page architecture on the other hand. Consecutive modifications are cached within a transaction-log which enables our architecture to work with respect to the well-known *ACID* paradigm. Besides the transaction-log as the writing-cache, we included caching in the *ReadTransactions* which results in the temporal storage of the actual page regarding the requesting transaction. Due to the binding of transactions to relevant *RevisionRootPages*, we minimize the caching of irrelevant data.

Based on the decoupling of representation and storage of the nodes, we are able to implement new node-types e.g. possible *CommentNodes* with only few effort. Such an extension results only in the adaption of the node-layer with no impact on the page- or transaction-layer.

Schema-aware modifications are possible as well. Due to the atomicity of insertions on node-level we have the ability to make constant checks against a registered schema. Even if this feature is not implemented yet, it would be easy to offer on-the-fly validation of modifications with respect to a schema valid for a defined set of versions. This would satisfy the temporal aspect of our approach without soften the validity of schema-based XML.

Besides checking for valid XML, our architecture offers great possibilities to support multithreaded modifications in the structure. Based on our encoding which only covers local areas of nodes and

our independent page-layer, concurrent *WriteTransactions* which are bound to concrete subtrees are possible as long as the modifications take place in disjunct subtrees. Since the page-layer is not encoding aware, it must be synchronized when it comes to concurrent modifications on the same *NodePages*.

Evaluation

Treetank is implemented in Java and an ongoing project within the Distributed System Group of the University of Konstanz. As a direct backend, Treetank works with flat files as well as with the Berkeley DB Java Edition[bdb] which is also used as the transaction-log for the *WriteTransactions*.

For the evaluation of our approach, we insert XMark[xma02] instances of two different sizes multiple times in our system. Each insertion results in a version. To keep the number of nodes per version constant, we remove the data from the old version as part of every new insertion operation.

Figure 7. Shredding and Serializing of XMark

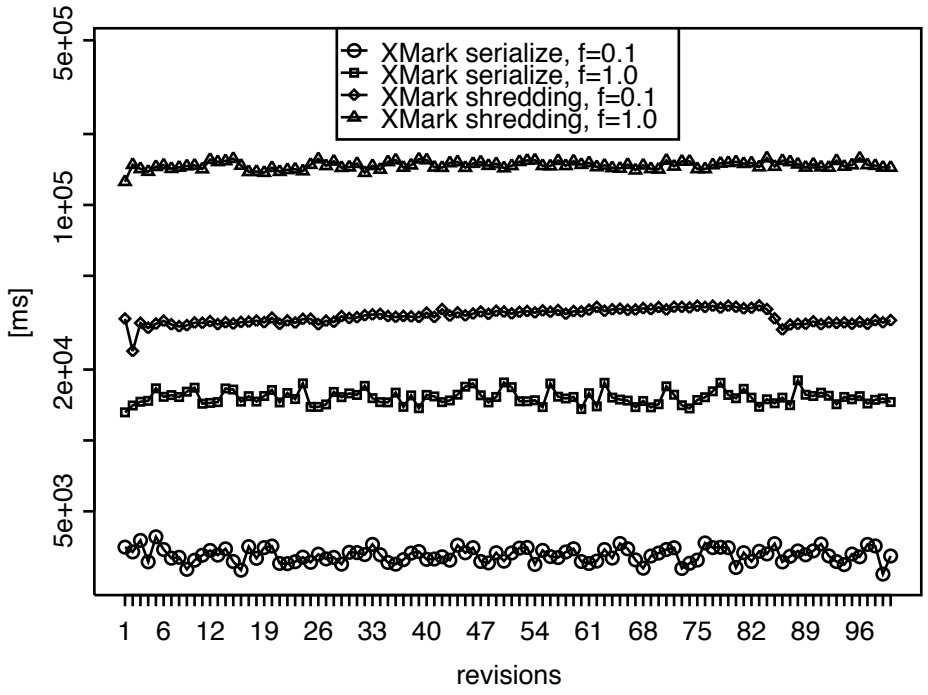


Figure 7 shows the result. The shredding represents the insertion process while the serializing stands for the retrieval process where the entire tree of one version is flushed to a simple *OutputStream*. We see that our systems scales with the size of the data as well as with the insertion and retrieval time. The write-operation takes always approximately the same time regardless of the version in which the data is inserted. The reason for this scaling regarding insertion time is our layered architecture which offers the same insert- and access time regardless of the requested version. Regarding the serialization of each version, the page-layer is touched in the same manner which offers similar retrieval performance within all versions.

The overhead of adaption becomes clearly visible within our random insert benchmark. *ElementNodes* are randomly inserted in the tree either as first-child or as right-sibling. After each insertion, a random move to a node is performed where the next insertion takes place. A commit is performed after the insertion of a fixed number of nodes (250, 500 and 1000). This operation continues until 1000 revisions are created.

Figure 8. Performing random insert, Time

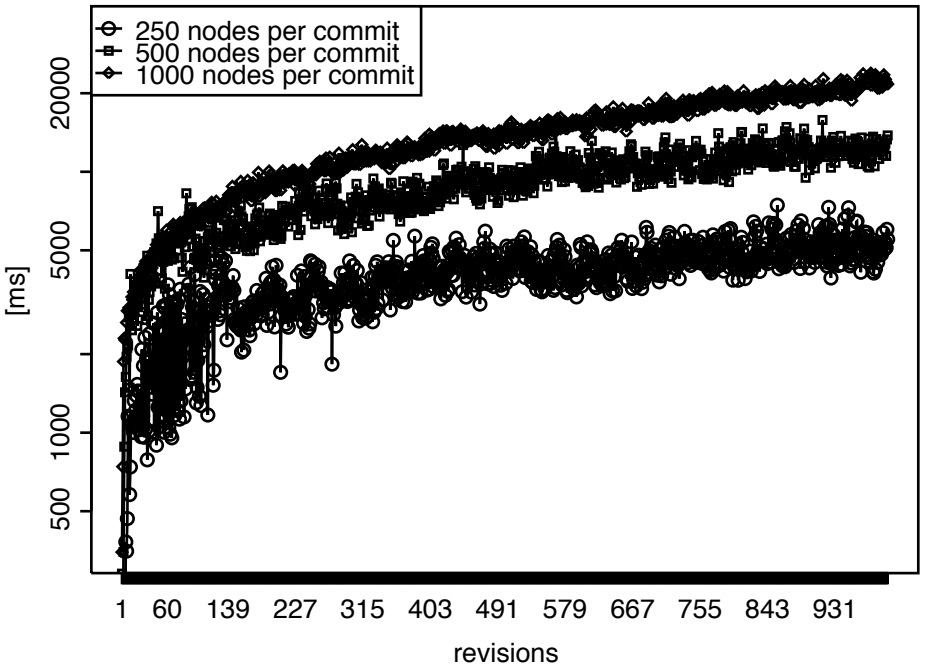
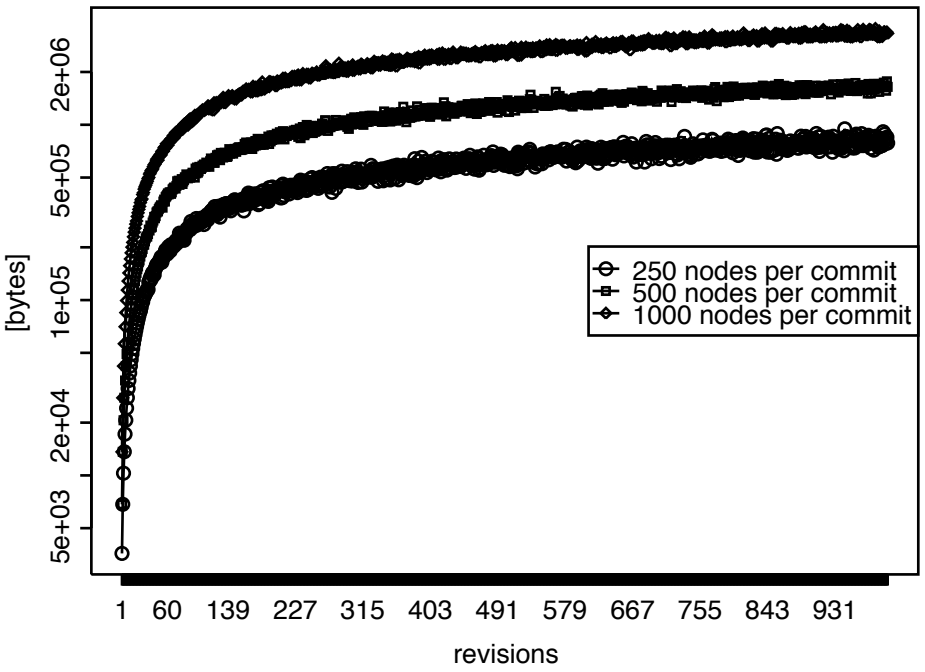


Figure 8 shows the insertion time. Since the insertion takes place on a constant increasing structure, the dereferencing of the sibling- and parent nodes for adaption and the storage of the new and modified pages needs logarithmic effort (since the y-axis is scaled in a logarithmic manner). This fits our architecture since we adapt only the neighborhood of a node and its ancestors for reconstructing the hashes. However, our system is getting more stable over time with an increasing number of versions.

Figure 9. Performing random insert, File



Regarding the storage size, we can rely at the moment only on relative comparisons, since compression and optimization of the backend were out of focus of our work so far. Nevertheless, the append-only approach results as well in a logarithmic adaption of the data regarding each incremental version. This is based on our page-layer which stores on the one hand all pages with nodes where the pointers have to be adapted and on the other hand the corresponding *IndirectPages* of new and modified *NodePages*. Since this random-insert operation results in huge deltas between two versions, logarithmic scaling of our storage satisfies our aim of a versioned storage even if we target to decrease absolut storage consumed by our approach in future.

Conclusion and Outlook

Treetank offers the combination of versioning and XML on node level. The key aspect of the proposed architecture are the different layers in our architecture which offers due to their independence from each other flexible adaptions to different workloads. Based on our specialized encoding and the append-only paging system, we are able to version any XML instance with respect to scalability and performance.

Furthermore, our node encoding offers multiple possibilities for extensions. Since we perform any operation on the nodes themselves, we plan to equip *Treetank* with integrity and security features like node-based permission control, XSLT-motivated information hiding and specific encryption operations for defined subtrees and versions.

The page-layer and its flexibility regarding versioning algorithms offer different extension points as well. Even if we already support multiple backends for the persistent storage, we plan to combine Treetank with our native Java iSCSI implementation called jSCSI. This would enable our architecture to store its data remotely based on block-based transmission. Another aspect of extension covers the optimization of the backend with respect to the absolut size of the storage. Furthermore, we develop a versioning approach which make read- and write-operations more predictable over time. This algorithm guarantees continuous read- and write loads without upcoming peaks and fits perfectly our dereferencing approach based on a set of multiple *RevisionRootPages* and the append-only paradigm.

Our transaction-layer will be extended as well. One of the upcoming features will cover concurrent modification accesses. These accesses are bound to fixed subtrees in which the operation will take place. Based on the locality of our approach, we can ensure that modifications bound to these subtrees are valid as long as the choice of the designated modification touches only disjunct subtrees. Upcoming security features on the node-layer will further affect possible other extensions of the transaction-layer.

Besides specific adaptions mapped to our layers, we further plan to use *Treetank* as native XML storage as well. Since our architecture allows concrete tracking of versions of huge XML instances, we develop visualizations for temporal trees. Furthermore we plan to extend our work in the field of distribution of XML and REST. These extensions will technically base on *Treetank* as well.

Bibliography

[bdb] *Berkeley DB*, `http://www.oracle.com/technetwork/database/berkeleydb/overview`.

[bdbxml] *Berkeley DB XML*, `http://www.oracle.com/technetwork/database/berkeleydb/overview/xquery-160889.html`.

[cvs] *CVS*, `http://savannah.nongnu.org/projects/cvs`.

[svn] *Apache Subversion*, `http://subversion.apache.org`.

[ord04] Patrick O'Neil, Elisabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. *ORDPATHs: insert-friendly XML node labels*. ACM SIGMOD International Conference on Management of Data. 2004. .

[tm10] Ghislain Fourny, Daniela Florescu, Donald Kossmann, and Markus Zacharioudakis. *A Time Machine for XML: PUL Composition*. XML Prague Conference. 2010. .

[tnt09] Vyacheslav Zholudev and Michael Kohlhase. *TNTBase: Versioned Storage for XML*. Balisage: The Markup Conference. 2009. .

[ubcc00] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. *Version Management of XML Documents*. The World Wide Web and Databases. 2000. .

[xma02] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. *XMark: a benchmark for XML data management*. International Conference on Very Large Data Bases. 2002. .

[zfs03] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. *The Zettabyte File System*. USENIX Conference on File and Storage Technologies (FAST). 2003. .