Evolutionary Tree-Structured Storage: Concepts, Interfaces, and Applications

Dissertation submitted for the degree of
Doctor of Natural Sciences

Presented by
Marc Yves Maria Kramis

at the

Universität Konstanz

Faculty of Sciences

Department of Computer and Information Science

Date of the oral examination: 22.04.2014
First supervisor: Prof. Dr. Marcel Waldvogel
Second supervisor: Prof. Dr. Marc Scholl

Abstract

Life is subdued to constant evolution. So is our data, be it in research, business or personal information management. From a natural, *evolutionary* perspective, our data evolves through a sequence of fine-granular modifications resulting in myriads of states, each describing our data at a given point in time. From a technical, *anti-evolutionary* perspective, mainly driven by technological and financial limitations, we treat the modifications as transient commands and only store the latest state of our data.

It is surprising that the current approach is to *ignore* the natural evolution and to willfully *forget* about the sequence of modifications and therefore the past state. Sticking to this approach causes all kinds of confusion, complexity, and performance issues. *Confusion*, because we still somehow want to retrieve past state but are not sure how. *Complexity*, because we must repeatedly work around our own obsolete approaches. *Performance issues*, because confusion times complexity hurts. It is *not* surprising, however, that intelligence agencies notoriously try to collect, store, and analyze what the broad public willfully forgets.

Significantly faster and cheaper random-access storage is the key driver for a paradigm shift towards *remembering* the sequence of modifications. We claim that (1) faster storage allows to efficiently and cleverly handle finer-granular modifications and (2) that mandatory versioning elegantly exposes past state, radically simplifies the applications, and effectively lays a solid foundation for backing up, distributing and scaling of our data. This work shows, using the example of tree-structured XML, that the characteristics and advantages of the *evolutionary* approach have been recognized and consistently implemented – something, which on its own is an important achievement.

We present the concepts of our *evolutionary* tree-structured storage TreeTank and the general-purpose SlidingSnapshot to prove that (3) formerly modification-averse tree encodings can be maintained with logarithmic update complexity, (4) linear read scalability beyond memory limitations is still guaranteed while maintaining logarithmic update characteristics, (5) secure copy-on-write semantics can be extended from the file level to the much finer-granular node level, (6) versioned node-level access is predictable and even realtime-capable, and, that (7) node-level snapshots are as or even more space efficient than page-level or file-level snapshots. In the course of our work, we inspired the Java-based iSCSI implementation jSCSI which proved that (8) high-level language block access is fast and also established the Java benchmark framework PERFIDIX as well as the block touch visualization tool VISIDEFIX.

We extend REST, the cornerstone interface of the web, with the ability to access the full version and modification history of a resource and call it (9) Temporal REST. This interface will not only encourage application developers to make use of our *evolutionary* approach, but it will also foster interactive and collaborative applications because they are, according to our claim (10), less complex to write and performing so well that users can now interactively work with large-scale data.

Finally, we provide an outlook on how evolutionary (full-text) indices, applications, and schemas can greatly leverage our contributions and how special-purpose hardware can speed-up our tree-structured storage while using far less energy. Especially our suggested approach to schema handling and evolution has the potential to radically simplify ORM-based software development.

Kurzfassung

Nicht nur das Leben, sondern auch unsere Daten sind einer beständigen Evolution unterworfen, sei es in Forschung, Industrie oder im Privaten. Aus einer natürlichen *evolutionären* Sicht entwickeln sich unsere Daten durch eine unablässige Reihe feingranularer Änderungen, die unzählige Versionen hervorbringen. Aus einer technischen *anti-evolutionären* Sicht, massgeblich durch technologische und finanzielle Einschränkungen entstanden, betrachten wir die Änderungen nur als vorübergehend und speichern vorwiegend nur die letzte Version unserer Daten.

Leider führt das Festhalten am gängigen Ansatz, die natürliche Evolution zu *ignorieren* und die vergangenen Versionen bewusst zu *vergessen*, zu Verwirrung, Komplexität und Geschwindigkeitseinbussen. *Verwirrung*, weil wir trotzdem immer wieder auf vergangene Versionen zugreifen müssen. *Komplexität*, weil wir wiederholt die Mängel unseres Ansatzes überwinden müssen. *Geschwindigkeitseinbussen*, weil Verwirrung gepaart mit Komplexität wenig Erfolg verspricht. Interessanterweise versuchen Nachrichtendienste notorisch, genau die Daten zu sammeln, zu speichern und auszuwerten, die die breite Öffentlichkeit bewusst verwirft.

Immer schnellere und günstigere Speicher sind der Haupttreiber für einen Wechsel hin zum *Nichtvergessen* vergangener Änderungen. Wir halten fest, dass (1) schnellere Speicher den effizienteren Umgang mit fein-granularen Änderungen sowie (2) einen eleganteren Zugriff auf vergangene Versionen ermöglichen, die Anwendungen vereinfachen und eine solide Grundlage für Backup und Verteilung unserer Daten legen. Die vorliegende Arbeit zeigt am Beispiel von XML, dass die Eigenschaften und Vorteile des *evolutionären* Ansatzes erkannt und konsequent umgesetzt wurden – eine Tatsache, die für sich allein eine wichtige Errungenschaft ist.

Wir beweisen an Hand unseres *evolutionären* baumstrukturierten Speichers TreeTank sowie des universalen SlidingSnapshot, dass (3) vormals änderungsaverse Kodierungen baumstrukturierter Daten in logarithmischer Zeit geändert werden können, dass (4) die lineare Skalierbarkeit lesender Zugriffe bei gleichzeitig logarithmischem Aufwand für Änderungen sichergestellt bleibt, dass (5) das sichere Kopieren-beim-Schreiben von der Datei- auf die wesentlich feiner-granulare Knoten-Ebene angewendet werden kann, dass (6) der versionierte Zugriff auf Knoten-Ebene vorhersagbar und echtzeitfähig ist, und dass (7) Snapshots auf Knoten-Ebene maximal so viel, oft weniger Platz benötigen, wie Snapshots auf Datei- oder Seiten-Ebene. Wir haben zudem die Entwicklung einer Java-basierten iSCSI Implementation namens jSCSI initiiert, an Hand derer gezeigt werden konnte, dass (8) Hochsprachen einen schnellen Zugriff auf Block-orientierte Speicher ermöglichen und haben zudem das Java Benchmark Framework PERFIDIX sowie das Tool VISIDEFIX zur Visualisierung von Block-Zugriffen etabliert.

Wir erweitern REST, die Kern-Schnittstelle des Internet, (9) um den Zugriff auf die volle Versions- und Änderungshistorie einer Ressource. Temporal REST wird interaktive Anwendungen beflügeln, weil diese, dank unserer Schnittstellen-Erweiterung (10) weniger komplex und so performant in der Ausführung sind, dass Benutzer interaktiv mit grossen Datenmengen arbeiten können.

Schliesslich zeigen wir auf, wie künftig evolutionäre (Volltext-)Indizes, Anwendungen und Schemas von unseren Beiträgen profitieren und wie spezialisierte, energiesparende Hardware unseren baumstrukturierten Speicher beschleunigen kann. Insbesondere unsere Anregung zur Arbeit und Evolution an und von Schemas hat das Potential, die ORM-basierte Softwareentwicklung radikal zu vereinfachen.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Prof. Dr. Marcel Waldvogel, for introducing me into the world of research and academia, both during my master and doctoral thesis. He allowed for a great amount of freedom to pursue my own visions and was at all times available with acute reviews and feedbacks. He was never tired to find new funding for my work and conference presentations. I'm really thankful that he sticked by me during my long lasting break due to family matters and even amicably warned me that writing a doctoral thesis while building up an own family and company would be hard – something which turned out to be so true.

The whole distributed systems research and computing center group warmly welcomed and sheltered me for almost five years. I received so much support and friendliness from Sabine Dietrich, Sylvia Pietzko, Stephan Pietzko, Gerhard Schreiner, Michael Längle, Peter Degner, Andreas Kalkbrenner, Jörg Vreemann, and Dr. Arshad Islam, among others.

I would also like to express my thankfulness to my second advisor, Prof. Dr. Marc Scholl for his valuable input and cooperation with his databases and information systems research group. I spent so many hours with Dr. Alexander Holupirek, Dr. Christian Grün, and Dr. Stefan Klinger, discussing new ideas, visions, and how to write them down to convince academia that they are worthwile. Barbara Lüthke also spent hours and hours trying to bring our English texts into a presentable form.

In the course of my work, I had the great opportunity to collaborate, and extend my knowledge in different fields. Notably with Prof. Dr. Daniel A. Keim and Dr. Florian Mannsman in the field of data analysis and visualization, Universität Konstanz, Prof. Dr. Torsten Grust and Dr. Jens Teuber in the field of query parsing and optimization, Technische Universität München, Prof. Dr. Sara I. Fabrikant and Cedric Gabathuler in the field of Geographic Visual Analytics, Universität Zürich, as well as Prof. Dr. Burkhard Stiller and my brother, Thierry Kramis, in the field of network traffic analysis, Universität Zürich.

Many thanks go to Prof. Dr. Dietmar Saupe, German Research Foundation (DFG) under grant GK-1042, Explorative Analysis and Visualization of Large Information Spaces, Universität Konstanz, especially for his patience waiting for my contributions and progress reports during my long break.

Besides my research, I was also involved with teaching and advising bachelor and master theses as well as research assistant work. I thankfully remember plenty of fruitful discussions and new inputs from Alexander Onea, Bastian Lemke, Daniel Butnaru, Giorgios Giannakaras, Halldór Janetzko, Hannes Schwarz, Johannes Lichtenberger, Markus Majer, Tim Petrowsky, Tina Scherer, Volker Wildi, and Xuan Moc, among others. They worked hard to implement and proof some of my ideas, and to evolve them. A special thank goes to Dr. Sebastian Graf, who greatly evolved and evaluated the concepts of TreeTank and SlidingSnapshot with my scarce support at that time.

The biggest thank of all goes to my mother, and, in memoriam, to my father, who encouraged and backed me throughout my studies, and, most notably, to my wife Annelie, who reinforced me to finish this thesis. Her love and moral support proved invaluable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Life is subdued to constant evolution. So is our data, be it in research, business or personal information management. The everlasting human impulse to adapt to new situations, learn, improve and review leads to a perpetual growth and modification of our knowledge – knowledge which is, in the information age, split up in small chunks, digitized and managed in persistent data stores, i.e., file- and/or database systems. Notably, the evolution is not only ceaseless, but also fine granular by it's nature.

From a natural, *evolutionary* perspective, data stores evolve through a sequence of modifications constantly transforming the state of the data store into another. From a technical, *anti-evolutionary* perspective, mainly driven by technological and financial limitations, we treat the modifications as transient commands which result in a single, i.e., the latest or current state and overwrite or delete all past states. Shocking for historians but business as usual for computer scientists. In fact, we willfully lose huge amounts of information and the capability to reconstruct the sequence of modifications (and likely the reason for them) as well as the past states of our data.

1.1 Anti-Evolutionary Approach

This current *anti-evolutionary* approach has drastic consequences for all of us because we somehow *feel* that we must go back to a past state some day or the other.

File-Level Hell Most users end up here anyway. We have to manually or automatically split our knowledge into files. Then we have to manually or automatically generate all kinds of backups or "copies" of our files because we are afraid, amongst other threats, to loose data due to unintended, buggy modifications. First, we must decide, how to split our knowledge into files. Second, we must decide, where to store the files. Third, we must decide what to backup. Fourth, we must decide where to backup to. Fifth, we must decide how to backup. Sixth, we must decide how to organize all of our backups to be able to restore them after all. And then we mess it up because we forget our decisions, because the backup just got too bulky, or the restore too complex because we must restore everything to just get a tiny bit of old state.

Database-Level Versioning A bit better due to its finer granularity, but still tedious, and application-specific. We have to repeatedly reinvent the wheel by implementing better or worse algorithms how to safely remember database records from past states. Note that the same decisions must be taken as with File-Level Hell. And then we dump the database to a single file just for backup purposes.

Filesystem-Level Versioning A bit better due to its more generic approach, but still file-level and application-specific. We do no longer have to care about the implementation details and get generic tools to backup and restore file systems and files at our hands. However, we just get the whole file and have to figure out intra-file state and modifications.

1.2 Evolutionary Approach

At some point in time – surprised – we asked us, why we suffer from so many drawbacks while storage gets bigger, faster, and cheaper at an astonishing rate. We asked us why we do not start to think the *evolutionary* way, i.e., think modification-driven and more fine-granular?

Most of the following explanation report seems to be simple and elegant because the *evolutionary* data versioning approach pervades all concepts and thoughts down to the node level and is integrated as a mandatory basic service. Without this clever move, most of the relevant technical properties would have needed more complexity and effort. This work shows, using tree-structured XML as a mental gymnastic apparatus, that the characteristics and advantages of node-level evolutionary data versioning have been recognized and consistently implemented – something, which on its own is an important achievement.

1.3 Claims

This thesis addresses several topics, which are listed below and revisited in Chapter 7.

1. **Degree of Granularity**
   We show that faster random-access storage hardware allows for an ever smaller granularity of the stored data. As such, storing the evolution of our data including even the tiniest of the intermediate steps gets ridiculously cheap.

2. **Evolution of State**
   We show that mandatory versioning, i.e., storing the modifications transforming one state into the other beautifully simplifies the applications and lays a solid foundation for backing up, distributing, and scaling of a data storage in a time- and resource-efficient fashion.

3. ***Pre/Post* Tree Encoding with Logarithmic Update Complexity**
   We show that the *pre/post* tree encoding can be updated with logarithmic complexity $O(\log n)$ by using counted B+ trees. This is a significant improvement over the current $O(n)$ update complexity.

4. **Linear Read Scalability beyond Memory Limitations**
   We show that the *Parent/First Child/Left Sibling/Right Sibling* tree encoding linearly scales beyond memory limitations when applied to persistent storage while keeping logarithmic update complexity. This allows to store and query tree-structured data sets orders of magnitudes bigger and faster than currently feasible.

5. **Secure Node-Level Copy-on-Write**
   We show that the checksum-protected copy-on-write, a.k.a., the log-structured approach, can not only efficiently be applied to the file level but to the much finer-granular node level.

6. **Predictable Realtime Node-Level Access**
   We show that any past version, or the sequence of modifications resulting in that version, can be accessed at node level with constant, predictable costs satisfying realtime requirements. Current systems either have to store much more data to achieve this, either trade logarithmic read or write with linear read or write, or invest enormous computing resources.

7. **Space-Efficient Node-Level Snapshot**
   We show that node-level snapshots consume less or at most the equal amount of space as page-level snapshots while still holding the predictability claim.

8. **High-Level Language Block Access is Fast**
   We show that a high-level language implementation of a block-level protocol such as iSCSI can be on par or faster than a low-level language. This also benefits proof-of-concept implementations of new ideas because they can be done and evaluated faster.

9. **Temporal REST**
   We outline an elegant temporal extension to REST to generically access any version or past modification of a web resource.

10. **Improved Workflow for Geographic Visual Analytics**
    We show how to speed up interactive and collaborative applications in Geographic Visual Analytics by one third by eliminating a whole intermediate step.

1.4 Publications

With the exception of SlidingSnapshot, from which the concept is presented exclusively in this thesis and from which the evaluation is presented exclusively in the thesis of Sebastian Graf [Gra14], all of the work in this thesis has been peer-reviewed and published.

1. Chapter 2: The work on the background and related research was published as *Growing Persistent Trees into the 21st Century* [Kra08a].

2. Chapter 3: The work on the TreeTank specification was awarded German patent number *DE 10 2008 024 809 B3* [Kra08b].

3. Chapter 3: The work on the distributed TreeTank was published as *Distributing XML with Focus on Parallel Evaluation* [GKW08] together with Sebastian Graf, and Marcel Waldvogel. Because the research on the distribution

aspect was mainly conducted by Sebastian Graf for his dissertation, we just present a brief summary and do not look deeper at this in our dissertation.

4. Chapter 4: The work on the evaluation of the linear scalability of TreeTank and other native XML databases was published as *Pushing XPath Accelerator to its Limits* [GHK+06], together with Christian Grün, Alexander Holupirek, Marc H. Scholl, and Marcel Waldvogel.

5. Chapter 4: The work on the evaluation of the node-level granularity of the TreeTank implementation was published as *Treetank, Designing A Versioned XML Storage* [GKW11], together with Sebastian Graf. Note that Sebastian Graf initiated the publication based on our implementation.

6. Chapter 4: The work on the Java iSCSI initiator jSCSI was published as *jSCSI – A Java iSCSI Initiator* [KWL+07], together with Volker Wildi, Bastian Lemke, Sebastian Graf, Halldór Janetzko, and Marcel Waldvogel.

7. Chapter 4: The work on the Java benchmarking platform PERFIDIX was published as *PERFIDIX : a Generic Java Benchmarking Tool* [KOG07], together with Alexander Onea, and Sebastian Graf.

8. Chapter 4: The work on VISIDEFIX was published as *Interactive Poster: Exploring Block Access Patterns of Native XML Storage* [JKK+06], together with Halldór Janetzko, Daniel A. Keim, Florian Mansmann, and Marcel Waldvogel.

9. Chapter 5: The work on Temporal REST was published as *Temporal REST – How to really exploit XML* [GK08], together with Georgios Giannakaras.

10. Chapter 6: The work on interactive Geographic Visual Analytics was published as *Streamlined workflow for large-scale interactive geographic visual analytics* [KG08], together with Cedric Gabathuler.

11. Chapter 6: The work on collaborative Geographic Visual Analytics was published as *An XML-based Infrastructure to Enhance Geographic Visual Analytics* [KGFW09], together with Cedric Gabathuler, Sara I. Fabrikant, and Marcel Waldvogel.

1.5 Overview

This thesis is structured as follows. Chapter 2 introduces the hardware impact on data storage in the past and within the next years, it defines the idea of *Degree of Granularity* as well as *Evolution of State*, it introduces the need for mandatory versioning, and the related work in this area.

Chapter 3 explains the concepts of our evolutionary tree-structured storage TreeTank and space-efficient SlidingSnapshot which applies snapshots at the node level. Chapter 4 evaluates our implementation of TreeTank with respect to its linear scalability and node-level granularity and also introduces the tools jSCSI, PERFIDIX, and VISIDEFIX.

Chapter 5 presents a temporal extension for REST which provides convenient web-based access to versioned resources. Chapter 6 shows how applications such as interactive and collaborative Geographic Visual Analytics greatly benefit from our ideas. Chapter 7 concludes this thesis.

Chapter 2

Background

The days of mechanical disks are numbered. Being a handy fellow for sequential access for many years, poor average random access times notoriously cause disks to struggle when it comes to handling large sets of XML data. Ripping out the mechanics and its inherent seek delays is an absolute *must* to allow for efficient and effective operations on fine-grained XML trees or their modification. In this thesis, we describe TreeTank, a system which takes advantage of zero-delay seek of flash-based (when compared to mechanical) storage, both addressing the strengths and weaknesses of flash, yet still performing rather well on traditional disks. The switch to flash keenly motivates to shift from the *anti-evolutionary* "current state" paradigm towards remembering the *evolutionary* steps leading to this state. Not only does this simplify many applications, it also offers a huge potential when it comes to accessing web-based resources in a temporal fashion. Being tuned for zero-delay flash-based storage, TreeTank will be able to provide more features faster and with less memory requirements than traditional approaches.

Despite the reputation of XML as being bloated, slow, and inefficient, it established itself as a first-class citizen throughout the modern computer world. As it expands and is adopted for a growing number of document formats, people *do* actually value features such as the self-descriptiveness of XML, the data-before-schema approach, the rich toolset, and the universal interchangeability of XML including long-term archival. This justifies the immersion of XML as a native data type into many programming languages and databases. However, dealing with large disk-based sets of XML data starting as low as several hundred kilobytes, can – without hesitation – be described as tedious. Opening and saving an OpenDocument file for a tiny modification can easily be in the spell of seconds. A daily download of an XML dump of Wikipedia for performing offline modifications is hardly feasible – not at least because the XML dump itself takes much longer than a single day [Lic07]. To put it bluntly, as the English would for XML, this is 'Typical!'.

It is currently not possible to efficiently and effectively modify large disk-based sets of XML data. The lack of modification efficiency and effectiveness is deeply rooted in two restrictions imposed by traditional persistent storage. First, the average random access time is so excessive that data needs to be extensively clustered and stored sequentially. This leads to an inefficient mismatch between the fine-grained logical and the coarse-grained physical data model. Second, the capacity is so scarce that applications try to be conservative in their storage needs, making only the most essential of their data persistent – regrettably excluding modification history and

past states. This jeopardizes the effectiveness of the user's workflow due to an unnaturally skewed focus on the current coarse-grained state instead of the fine-grained modification history. However, the switch to flash-based storage does not only improve the situation, it also brings along its own problems: flash-based storage will eventually wear out if the blocks are overwritten too many times and the block erasure procedure consumes a significant amount of time [MD11].

From the bird's eye view, our contribution is two-fold and consists of a background analysis as well as a synthesis resulting in a tangible system named TreeTank.

Analysis We uncover the deficiencies of traditional storage and show how flash technology alleviates them. Average random access times are significantly shrinking with the advent of each new storage technology and due to its evolution over time. Simultaneously, capacities are increasing steadily. Consequently, we find a clear tendency from coarse-grained storage units such as flat files or binary large objects towards fine-grained record-, tree-based, or semi-structured storage which does not only store the current state but also the evolutionary steps leading to this state.

Synthesis Our system overcomes the traditional limitations by consistently tuning data structures for flash-based storage while still working with magnetic disks and lowering the memory requirements. TreeTank provides a scalable, lightweight, transactional, secure, and persistent framework for efficiently and effectively modifying fine-grained data structures such as XML.

2.1 Hardware Impact

Table 2.1 lists one state-of-the-art product for each major persistent storage technology in the order of appearance. This includes magnetic tape [Ora08], magnetic disk [Sea08], and flash [Fi08]. Volatile memory [AD08] is added for the purpose of comparison. The columns have the following meaning: type of storage, capacity, price per capacity (based on Internet research as of the time of writing), sustained sequential read throughput, average random access time, and mixed I/O operations with a queue depth of one and a size of 8k.

Type	**Capacity** [GB]	**Price** [$/GB]	**Sequential Read** [MB/s]	**Random Access** [s]	**Operations** [IOPS]
Tape	500	0.3	120	6.2E1	1.6E-2
Disk	73	6	96	2.9E-3	1.8E2
Flash	80	30	700	5.0E-5	8.8E4
Memory	2	35	7800	6.4E-8	1.0E6

Table 2.1: *Comparison of persistent storage with volatile memory*

The parameters indicate that each new technology brought persistent storage closer to volatile memory. In stark contrast to strong similarities of capacity, price per capacity, and sequential throughput within one order of magnitude, average random access time and input-output operations per second show a wide discrepancy by two to four orders of magnitude. In addition, each technology itself saw continuous enhancements. E.g., IBM introduced magnetic disks in 1955 with the model 350 Disk Storage Unit being a part of the IBM 304 RAMAC (Random Access Memory Accounting) [IBM55]. Disk capacity was about 4.8MB and memory

capacity about 2.3kB. Hence, the capacity of both disk and memory rose over four orders of magnitude within the better half of the last century. This trend is yet unbroken and close to six orders of magnitude when the focus is not performance but capacity.

Interestingly, the number of I/O operations per second was further improved by flash by truly parallelizing the access in analogy to the central processing units that do no longer only ameliorate the performance by making a single core faster but also by adding more cores.

2.2 Degree of Granularity

The most stringent limitation of mechanical disks is their ropy average random access time. Given both a fixed amount of data and time, average random access time determines the number of I/O operations per second as well as the size of the moved data. The higher the I/O operations per second, the more data objects of smaller size can be shuffled around. In other words, average random access time has an immediate effect on the granularity at which data objects can be handled efficiently. From a conceptual perspective, tapes work best at file-level granularity. Disks can deal with record-level granularity. Flash pushes granularity to the field or node level. Memory eventually is the candidate of choice when it comes to byte-level data processing. Figure 2.1 gives a conceptual illustration of the relationship between Degree of Granularity, average random access time, and object size.

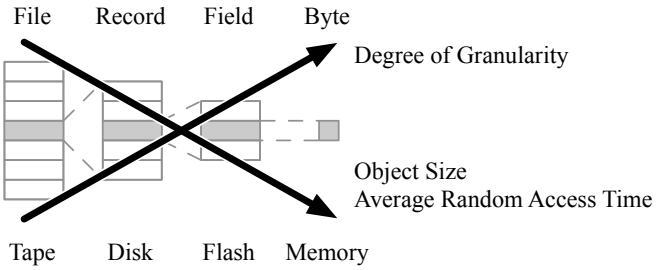


Figure 2.1: *Degree of Granularity in relation to average random access time and object size*

The only way to work at a finer granularity than available with a given storage technology, is to switch to sequential processing or to temporarily store all or a part of the data with a technology that allows a finer granularity. Talking about XML, which is a fine-grained *unranked ordered* tree, it immediately becomes clear why there must be a penalty with traditional disk-based storage.

Theoretical Penalty The logic of *unranked ordered* trees was thoroughly analyzed in [Lib06]. It has been shown that the *unranked* and *ordered* properties have a significant impact on the runtime characteristics, automata models as well as temporal and modal logics of a tree compared to the simpler *ranked* and *unordered* properties. Note that *unranked* means that the number of children of any node is *not* limited and that *ordered* means that the children of all nodes are ordered by *sibling ordering*. Besides this, the node order must be defined and sequentialized by a depth-first (also known as *preorder*) traversal of the tree assigning a steadily increasing number to each node starting from 1 for the root node.

Practical Penalty XML must be stored sequentially. For random node-level access, it must first be parsed into memory. Once all XML nodes are in memory, they can be randomly accessed and modified. If a modification takes place, all nodes must be sequentially serialized back to disk. As such, the mismatch between XML's fine and the disk's coarse granularity consequently leads to a loss of efficiency when it comes to random access or modifications.

Things change considerably when taking flash into account. With the finer granularity, each XML node is directly accessible by its key or position in the XML tree. The requirement to physically cluster related nodes can be dropped. Sequentially accessing physically dispersed nodes on flash-based storage will be in the same order of magnitude as accessing physically clustered nodes on a disk. As a side effect, memory can be used much more efficiently to just cache the frequently used nodes instead of caching all nodes.

The evolution of persistent data structures backs our observation. In the early days, merge sort was the prevalent method to keep data organized on tapes. Now, while merge sort is still a valuable topic to teach and now and then appears in practice, B+ trees or even hash storage dominate the field. The practical implication of this development is impressive. Tape-based systems frequently run merge sort to avoid data fragmentation due to insertions or deletions. Disk-based systems intermittently de-fragment their file system trees for the same purpose. In stark contrast, flash-based systems are indifferent as the performance does not degrade with scattered data. Generally speaking, shorter average random access time leads to less management overhead due to data fragmentation.

2.3 Evolution of State

Each modification evolves an existing state into a new one. Both the modification and the state are bound to a given point in time. The current state is an aggregate of all past modifications taking place during a given time period. Often, the modification is small compared to the new state it creates. Again, we find a mismatch between fine-grained modification and coarse-grained state. Figure 2.2 illustrates the relationship between the state and its evolution.

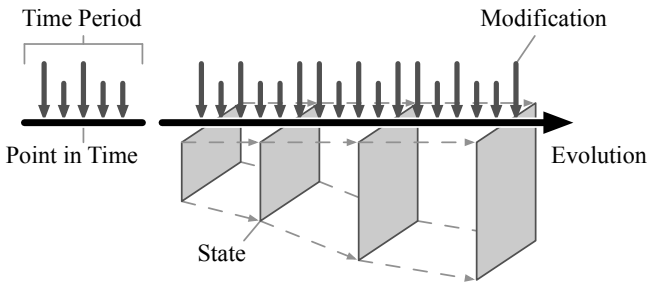


Figure 2.2: *Evolution of State through a sequence of modifications*

Given the constraint to cluster data due to poor random access performance, the system can either try to cluster modified data in-place by overwriting old data or out-of-place by writing it to a free place. Given the constraint to overwrite clustered data due to the limited capacity, out-of-place traditionally is only a choice when the old place is marked as free. Looking back to our OpenDocument example, it

becomes clear why a user cannot effectively modify even a medium-sized XML file. Before the modification, the XML is sequentially read into memory. Now the user makes a small modification by, say, adding a new XML node representing a section title in the middle of the XML node tree. Instead of making this tiny modification persistent, the whole XML must be sequentially written back to disk overwriting the old XML file. Both the old file and the modification are lost. What remains is the current state.

The fact that the system only knows the latest, i.e., current, state is widely accepted as the user sees the result of her modification. However, big efforts are required as soon as the user wants to do backup, undo, or redo operations. As for the backup, this can currently only be achieved manually by the user or with a separate application that laboriously determines the difference between the last backup and the current state. Once the difference is detected, an incremental delta is backed up. As for undo or redo operations, they currently either only span a single session between the opening and closing of the document or they have to be stored as a modification history together with the actual data. It would be much easier if the system inherently knew about the modification history and would be able to persistently reproduce the state after each modification.

Flash is well suited to model fine-grained modifications. The system gains the freedom to decide whether it should make the modification or the state resulting from this modification persistent. The former allows to quickly answer questions about the modification history, the latter to swiftly reconstruct the state at a given point in time. E.g., the insertion of the section title can be achieved by simply storing the remark about the inserted XML node or by storing the whole sub-tree as it looked like after the insertion. The amount of data written is negligible for the modification-only when compared to the whole-state variant. As a side-effect to the modification-only variant, the backup application could ask for the last modification and just backup the freshly inserted XML node. Furthermore, the user could decide right after the next start of her application whether she would like to undo the new section title.

Interestingly, the freedom of choice either to store the modifications or the state is a burden on its own. Storing the modifications needs efforts to reconstruct the state. Storing the state needs efforts to reconstruct the modifications. The related work shows more or less efficient ways how to bridge this gap between the two approaches. We will show in Chapter 3 how to cleverly balance and satisfy both requirements at once, i.e., how to efficiently store and retrieve both the modifications and the state in a single concept.

2.4 Related Work

In Section 2.2 and Section 2.3, we analyzed the limitations of traditional persistent storage and assumed a traditional file system as the intermediary between XML and storage. In this subsection, we analyze advanced systems which employ sophisticated data structures to push the limits set by magnetic disks. We investigate related work in the area of file, database, versioning, XML, and distributed systems as well as recent combinations thereof. All of these systems largely depend on persistent storage. We look at its impact on how each system organizes data.

2.4.1 File Systems

We perceive six major developments in the area of file systems. First, transactional object store. Second, copy-on-write. Third, end-to-end integrity. Fourth, file system event messaging. Fifth, full text index. Sixth, event-based backup.

While the transactional object store (also known as Data Management Unit, DMU) of Sun's ZFS [BAH+03, ZFS04] currently works at file-level granularity, it is readily available for storing finer-grained objects. ZFS is one of the first widespread file systems to integrate transactional behavior on the basis of a single write transaction combined with concurrent reads. The copy-on-write in ZFS is a mix of a log-structured [RO92] and a traditional file system. The former only appends data and the latter only overwrites data. ZFS write transactions append data. But in case the user does not mark it as a long-term snapshot, the freshly appended data eventually gets overwritten to save capacity. The end-to-end integrity is an important feature as it can deal with many failure scenarios uncovered with simple error detection codes on various underlying layers. It even allows for cryptographic-strength integrity checks and is a tribute to the ever growing capacity as each hash or message authentication code [Fed02b] allocates up to several dozen bytes.

Apple introduced file system event messaging, full text index, and event-based backup with FSEvents [Ars07a], Spotlight [Sin06], and Time Machine [Ars07b] respectively. To make the best out of these new technologies, Apple pragmatically splits large files into many small files. E.g., a single file which used to store multiple mails or events is split into multiple files, each storing a single mail or event. Finer semantic data granularity leads to a higher precision when it comes to communicate file modifications to applications such as the full text index Spotlight – another tribute to shrinking average random access times. Notably, the modification events are made persistent and aggregated gradually not to waste too much capacity. A notorious user of the novel event messaging framework is Time Machine. Instead of intermittently searching and calculating the deltas between the last backup and the current state to perform an incremental backup, Time Machine asynchronously consumes fine-grained modification events and only backups the affected files. Note that still whole files are backed up but since the files get smaller, the granularity is finer as when compared to other systems.

2.4.2 Database Systems

With database systems, we find an interesting approach with Berkeley DB Java Edition which stores a traditional B+ tree in a log-structured fashion [Ber91]. B+ trees play an important role in database systems. They allow to keep the data sorted and to quickly retrieve it both sequentially and randomly. However, the data must still be clustered not to over-stretch the capabilities of traditional storage and the tree must be kept balanced after modifications. Like ZFS, Berkeley DB clusters writes and appends them sequentially. Berkeley DB does not store the modifications and is just able to reconstruct the last successfully committed state. Furthermore, it has to reconstruct the B+ tree in-memory to provide reasonable random access to the physically scattered data.

Recent work tries to tackle the comparably long write time of flash when the write occurs for the second time due to the block erasure procedure. FlashDB [NK07] tries to self-tune its B+ tree by analyzing the workload and switching between a disk, i.e., random, and a log-structured mode. In-page logging (IPL) [LM07] tries

to take advantage of both the traditional in-place and the log-structured approach. IPL tries to minimize the number of block erasures by reserving a small space in each page for future modifications. Only if the reserved space is consumed, the page must be written to a new location – potentially involving a block erasure.

2.4.3 Versioning Systems

Versioning systems are closely related to backup systems. However, versioning systems focus on the *workflow* requiring frequent access to past versions as well as the modification history while backup systems focus on the *availability* in the event of user, software, or hardware failures. Both have in common that they currently either use a full, differential, or incremental approach [Lic07] on how to store and retrieve past modifications or state as compared in Table 2.2.

Type	Storage Consumption	Write Time	Retrieve State	Retrieve Modifications
Full	−	−	+	−
Differential	=	=	=	=
Incremental	+	+	−	+

Table 2.2: *Comparison of the main approaches on how to version data*

Full Storing the whole state after a set of modifications is called full versioning. It is the most space consuming approach, causes the most effort to retrieve past modifications but is the fastest way to retrieve past state.

Differential As an improvement over full versioning, differential versioning only stores the state difference (not the modifications) after a set of modifications compared to the last full state. As such, it could also be described as storing the cumulative state increments. This reduces storage consumption and time requirements but degrades over time because the state difference steadily grows. Intermittent full states are still required.

Incremental As an improvement over differential versioning, incremental versioning only stores the state difference (not the modifications) since the last state. As such, it could also be described as storing the differential increments only. This further reduces storage consumption but takes longer to reconstruct a past state. Intermittent full states are still required.

One of the earliest approach to professionally work with versions was established in the area of source code version control. Systems such as CVS [CVS89] or its successors SVN [Apa00] or GIT [Swi08], among others, fostered collaborative source code authoring, initially in a centralized (CVS, SVN) or now also in a distributed (GIT) fashion. Notably, these versioning systems focus on the file level and leave it to the user and her tools to find out about intra-file changes.

Revlog [Mac06] is an important contribution in the area of differential versioning systems. Revlog stores deltas of files which change during a version. Each delta is derived by a diff algorithm comparing the last file version with the current one. Intermittently, a full snapshot is stored to accelerate the retrieval of a file in a given version. Without snapshots, Revlog would have to reconstruct the original file by sequentially applying all intermediary deltas up to the requested version, starting from the current one. As such, Revlog allows to efficiently keep a version history

of all deltas. Both the state of a file as well as the modification evolving this state are derivable with reasonable effort. Still, the diff algorithm is time-consuming and not aware of the modification semantics.

Etherpad [GIZ08] is a special incremental versioning system, because it uses semantic modifications and stores them as events. It is one of the few fine-granular systems strictly adhering to the *evolutionary* approach. To increase the efficiency, groups of events are hierarchically aggregated and stored as a single event. This aggregation process is very efficient because many modifications are sequential or at least share a spatial locality. However, it only works with a limited amount of data and in a special collaborative text editing environment.

2.4.4 XML Systems

XML systems are also known as (native) XML (database) systems. Interestingly, the average user still stores most XML as flat files instead of one of these optimized alternatives. We distinguish four approaches by how XML systems locate a node in the fine-grained unranked ordered XML tree. First, fixed-size key. Second, variable-size key. Third, positional key. Fourth, index-based key. Table 2.3 lists the four types and compares key stability, global order, whether it is extractive, and random write performance.

Type	Key Stability	Global Order	Extractive	Random Write Performance
Fixed	yes	no	yes	++
Variable	yes	yes	yes	+
Positional	no	yes	yes	−
Index-Based	no	yes	no	−−

Table 2.3: *Comparison of the main approaches on how to store XML with fine-grained data structures*

Fixed-Size Key Persistent DOM [HMF99] is an example for fixed-size keys. Each node is located by a unique immutable key of fixed size. This key does not necessarily reflect global order and must be stored. Modifications are efficient because they involve at most the modified, parent, left sibling, and right sibling node. However, updates do not scale when a node contains a large number of child references. Reads on the other hand are only efficient if the global order of two nodes must not be established.

Variable-Size Key ORDPATH [OOP+04] is an example for variable-size keys. Each node is located by a unique immutable key of variable size. This key maintains global order and must be stored. The key is derived during node insertion depending on the location of the node in the tree. Hot-spots seeing frequent node insertions within the same sub-trees lead to long keys and thus restrict write scalability. Sequential and random reads can be done efficiently unless long keys are involved.

Positional Key XPathAccelerator [Gru02] is an example for positional keys. Each node is located by its unique mutable position in the tree. The position respects global order and is not directly stored. Writes are efficient with a positional B+ tree [GHK+06]. However, the usual implementations have a limited update capability which is only achieved by leaving gaps in the

positional numbering. Reads are scalable as long as only forward axes are involved. BaseX [GHS07] is a special implementation that shrinks the size of each node by further dropping support for the preceding axis. Tightly packing XML nodes allows to store more of them in-memory and to notably accelerate processing.

Index-Based Key Virtual Token Descriptor (VTD) [Xim04] is an example for index-based keys. VTD is the only non-extractive system as it does not extract strings but directly references them at their position in the original XML file. Each reference is of fixed size and equivalent to an index key. Write scalability is provided, as long as VTD can sequentially process XML files. Random insertions or deletions still require to re-serialize both the XML file and the index. As such it is a close relative of the positional key approach. Reads are efficient for sequential access. Random access must be supported with the help of in-memory location caches linking parents and their respective first child.

2.4.5 Distributed Systems

Research started to look into distributed XML data only a few years ago. Many approaches considering distributed queries are based on the assumption that XML is already distributed [Suc02, BF05, BCFK06, ABC+03, BG03]. The focus is laying on the distributed query evaluation itself. Based on the well-known distribution techniques of relational databases, i.e., the horizontal [CNP82] and vertical fragmentation [NCWD84], some take this straightforward concept of fragmentation into account [MS03, LZS+02, MS05]. The suggested algorithms work well for data-oriented XML because of their regular structure.

Based on document-centric XML, the resulting XML fragments could have different structural characteristics. To our knowledge, there are only a few approaches, which take the structure itself into account to avoid possible irregularities when partitioning and distributing an XML-tree. [BC07] presented an approach which is directly based on several structural constraints. i.e. the width, the size, and the depth of sub-trees which can be extracted. In addition, the parameters have to be manually set before-hand to obtain a fragmentation. Depending on these parameters, a good fragmentation with respect to a parallel evaluation is guaranteed.

A completely different approach with the same focus on parallel queries is described in [LCP06]. The parallel evaluation takes place either on distributed XML which was partitioned with the help of graph-partitioning algorithms [KVK99] or on a variable fragmentation depending on an executed query. In this case, the fragments are represented by DOMs. This reduces the usability of the variable fragmentation because the DOMs have to be adapted each time the query changes.

2.4.6 System Convergence

Based on the traditional file, database, versioning, XML, and distributed systems, recent research suggests combinations thereof. Typically, the versioning or the distribution aspect is integrated with a traditional file or XML system. Zholudev and Kohlhase presented TNTBase, a combination of an XML and a versioning system [ZK09]. TNTBase builds on top of SVN [Apa00] and Berkeley DB XML [Ber03]. However, they still handle XML at the file and not at the node level.

Another approach to XML versioning systems is the Time Machine for XML [FFKZ10]. It represents the deltas between XML versions as XQuery PULs and stores the versioned nodes in a data structure called pi-tree. The ORDPATH encoding [OOP+04] is used and requires the underlying page architecture to support clustering to overcome the linear search for the suitable pi-nodes of a given revision. UBCC [CTZ00] tries to overcome this limitation by introducing page thresholds. If a predefined threshold is reached, the pages are rearranged regarding their contents. Unfortunately, this reorganization can result in peaks and synchronization issues related to the read-/write-performance.

A recent publication [MBHM13] suggests ORI as a distributed versioning file system. It combines the features of a file system with both, versioning and distribution, and confirms the clear trend towards our *evolutionary* approach. While the authors of ORI question the decades-old file system interface and agree on the importance of versioning, they still do not extend their ideas to the more fine-granular intra-file level, as, e.g., Alexander Holupirek [Hol12].

2.5 Summary

Our background analysis of related work shows a trend towards finer granularity from the semantic and the storage perspective. In addition, more and more systems try to introduce some notion of evolution and past state complementing the prevalent *anti-evolutionary* current-state-only philosophy. Consequently, all systems challenge the limits given by average random access time as well as capacity and closely follow the technological development towards faster random access and growing capacity. Nevertheless, most systems still assume mechanical disks as their underlying storage which leaves room for further improvements when designing for flash as shown, e.g., by FlashDB [NK07] or IPL [LM07].

We analyzed why it is currently not possible to efficiently and effectively modify large disk-based sets of XML data. We identified traditionally poor average random access times of mechanical disks as a major problem. Flash-based storage smashes this technological hurdle twofold. First, it prepares the ground to align the Degree of Granularity of logical and physical data models to enhance efficiency. Second, it allows to store fine-grained modifications instead of coarse-grained state to improve the effectiveness of the user's workflow. An overview of state-of-the-art file, database, versioning, XML, and distributed systems as well as recent combinations thereof, shows the trend towards finer-grained data structures to better model user requirements. However, the trend is bound by technological progress of mechanical disks and does not yet consider flash as its underlying storage.

We suggest TreeTank as a system to take full advantage of flash-based storage while not dropping support for erstwhile mechanical disks. TreeTank enables node-level access faster than traditional systems and without their extensive memory requirements. TreeTank provides a scalable, lightweight, transactional, secure, and persistent framework facile to implement, dependable to run, and modest to maintain. Additionally, we suggest SlidingSnapshot, which endows the user with the freedom to query both the node state for a given version as well as the node modification history between two versions. A detailed specification, implementation, and evaluation of TreeTank and SlidingSnapshot can be found in Chapter 3, Chapter 4, and, with respect to the evaluation of SlidingSnapshot, in [Gra14].

Chapter 3

Concepts

In this chapter, we present TreeTank as a unified storage manager concept for *evolutionary*, tree (but not limited to) data structures such as B+ trees or tries. It is compliant with an interweaved set of features, which are:

- Protected transactional access which allows for multiple parallel read and (for now) a single write transaction (see Subsection 3.1.1).
- Highly parallel multi-core-capable architecture well suited for software and hardware implementations (see Subsection 3.1.1).
- Integrated node-level access to all past modifications and states of the stored tree (see Subsection 3.1.2).
- Time-proven security algorithms for strong encryption and end-to-end integrity (see Subsection 3.1.3).
- Optimized on-device storage layout for best-performing concurrent random read and sequential write on flash-based storage (see Subsection 3.1.5).
- Preparation for solid, automatic, and incremental backup on a single or multiple local or remote storage managers for active usage of redundancy (see Subsection 3.1.5).
- Greatly improved space efficiency for modifications thanks to concepts such as SlidingSnapshot and dynamic page compression (see Section 3.2 and Figure 3.2).

We also give a short description of the general-purpose concept of SlidingSnapshot which allows for space-efficient, node-level, and realtime-predictable access. The concepts of TreeTank described in Section 3.1 and SlidingSnapshot described in Section 3.2 can mutually benefit from each other, but can also be used separately. Finally, we motivate the distribution of our concepts in Section 3.3 because this will allow for inherent scalability, parallel processing, and availability.

3.1 TreeTank

TreeTank stores all versions of an unranked ordered tree in a set of pages. Each page stores a set of page references pointing to other pages as well as a set of nodes containing the application-specific data. From a physical perspective, TreeTank stores the per-page and per-version modifications as page deltas. Note that a delta is not the result of an expensive diff calculation but just the plain modification event. Intermittently, a full page snapshot is stored for each page to fast-track its in-memory reconstruction. Consequently, TreeTank can quickly derive the state of each node in each version as well as the modification history of each node between two versions. Note that SlidingSnapshot could be used instead of the traditional intermittent full snapshot algorithm.

TreeTank was designed with security in mind. This involves the security primitives authentication, confidentiality, integrity, non-repudiation, access control, and availability. According to Schneier [FS03], the user is only left with one option, i.e., whether security is turned on or off. If activated, a small set of secure, fast, and time-proven algorithms is used: CTR-AES-256 [NIS01a, Fed01] for encryption, SHA-256 [Fed02a] for key salting and stretching, and HMAC-SHA-256 [Fed02b, Fed02a] for authentication.

Each instance of TreeTank is bound to a session. The session allows a single write and multiple concurrent read transactions at any time. The write transaction is bound to the latest successfully committed version and allows to modify it in-memory. A new version is created and all modifications are serialized sequentially when the write transaction commits. Each read transaction is bound to a committed version and allows to read the page tree in this version.

TreeTank stores all data and metadata on the primary logical device. The secondary logical device just contains replicated metadata for safety and performance reasons. Both logical devices may grow by appending more sectors. To prevent wear-out of the flash device, data is only appended. To provide optimal write performance, data is only written sequentially. The header contains the configuration data and is replicated four times. The version reference pointing to a version is replicated twice. The page snapshots and deltas are stored once. The replication to additional local or remote devices is trivial and optimally performing because it sequentially works on the block-level with constant search time for the first block to start with.

Binary search is used twice with TreeTank. First, it finds the last successfully committed version. Second, it finds the closest version number for a given point in time. In both cases, binary search works on the array of version references stored on the primary logical device. TreeTank guarantees that at least one version reference exists. A version reference is valid if the first eight bytes are not zero. To find the last successfully committed version, binary search looks for the right-most valid version reference. With each chosen median, the binary search continues to the right, if the version reference is valid, else, it continues to the left. To find the closest version number for a given point in time, binary search asserts the validity of each chosen version reference and then compares the provided point in time with the stored one. The search finishes, if either an exact match was found or the smallest possible time difference.

In a nutshell, B+ trees always cluster data within each page of the tree. TreeTank only clusters data during snapshots and usually just stores deltas. This

trades random access time for space and availability of the full modification history. E.g., a rough approximation (calculations are based on Table 2.1) shows that a magnetic-disk-based B+ tree with five levels requires 14.5ms to find a data item. A flash-based TreeTank with five levels and ten deltas per level on average requires 2.5ms to find the same data item. TreeTank can tune the snapshot frequency to adapt itself to the available storage and workload. Furthermore, it does not depend on in-memory caches to speed-up its operation.

The tree encoding uses the update-friendly *Parent/First Child/Left Sibling/Right Sibling* tree encoding as depicted in Figure 3.1. Also note that we use the acronyms as listed in Table 3.1.

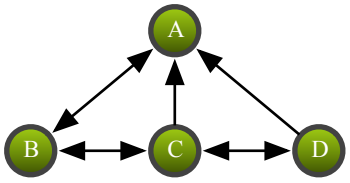


Figure 3.1: *Encoding of the unranked ordered tree. The parent node A has a reference to its first child B. The children B, C, and D have a reference to their parent A as well as to their immediate left and right siblings*

3.1.1 Sessions and Transactions

Each instance of TreeTank is bound to a session *SN* that controls read and write access. *SN* allows a single write transaction *WTX* and multiple concurrent read transactions *RTX* at any time. Each transaction is run by a dedicated thread. Both *WTX* and *RTX* are started from *SN* and must be closed before closing *SN*.

WTX allows to modify the underlying TreeTank, i.e., to insert, overwrite, and delete nodes starting from the last successfully committed revision. All modifications are made to in-memory logical pages exclusive to *WTX*. *WTX* must either be committed or aborted before closing. An abort drops all modified in-memory logical pages. A commit serializes all in-memory logical pages to the corresponding physical fragments. The process of serializing an in-memory logical page to the corresponding physical fragment is depicted in Figure 3.2.

WTX can be committed by the user at any time or at system-defined times. The system will commit based on how many node modifications occurred, the elapsed time since the last commit, or on memory pressure. *RTX* allows to read the underlying TreeTank as it was in any revision up to the last successfully committed revision. Modifications are not allowed. All *RTX* have access to a shared cache of in-memory logical pages.

3.1.2 Revisions and Pages

From a logical perspective, revision R_r consists of a tree of logical pages. A deterministic path p leads to a logical page $P_{r,p}$ irrespective of r. All possible p are enumerated starting at 0 which stands for the deterministic path to the logical root page of the logical page tree, i.e., $P_{r,0}$. $P_{r,0}$ is found through the revision reference RR_r.

Acronym	Description
CNF	System-dependent configuration of type `Byte[448]`
CTR	Counter of type `Int[2]`
H	Header
HT	Header token of type `Int[8]` defined by HMAC-SHA-256(K, *CNF*)
$INP_{r,p}$	Indirect $P_{r,p}$
K	Symmetric key of type `Int[8]` derived from *MK* and SLT_h through stretching and salting
LD_d	Logical device d
MK	Symmetric master key of type `Int[8]` randomly chosen by the user
N	Nonce of type `Int[2]`
$ND_{r,p,o}$	Node at offset o of $NDA_{r,p}$
$NDA_{r,p}$	Node array of $P_{r,p}$
$NDL_{r,p}$	Node list of $PS_{r,p}$
$NDLS_{r,p}$	Size of $NDL_{r,p}$
$NDP_{r,p}$	Node $P_{r,p}$
NEXDB	Native embedded XML database
$NMP_{r,p}$	Name $P_{r,p}$
$PD_{r,p}$	Variable-length page delta of $P_{r,p}$
$P_{r,p}$	Page p of R_r
$PP_{r,p}$	Page part p of R_r which is either a $PS_{r,p}$ or a $PD_{r,p}$
$PPC_{r,p}$	1B page part count of $PR_{r,p}$
$PPR_{r,p}$	32B page part reference of $PP_{r,p}$
$PPL_{r,p}$	2B page part length of $PP_{r,p}$ in words
$PPO_{r,p}$	6B page part offset of $PP_{r,p}$ in words
$PPT_{r,p}$	24B token of $PP_{r,p}$ defined by HMAC-SHA-256(K, $PP_{r,p}$)
$PR_{r,p,o}$	Page reference at offset o of $PRA_{r,p}$
$PRA_{r,p}$	Page reference array of $P_{r,p}$
$PRAO_{r,p}$	Offset of $PR_{r,p}$ in $PRA_{r,p}$
$PRL_{r,p}$	Page reference list of $PS_{r,p}$
$PRLS_{r,p}$	Size of $PRL_{r,p}$
$PS_{r,p}$	Variable-length page snapshot of $P_{r,p}$
R_r	Revision r
$RD_{r,p}$	Revision difference between $PP_{r,p}$ and its preceding $PP_{r',p'}$
RN	Root node
RR_r	Revision reference pointing to $PS_{r,0}$
RRT_r	32B token of RR_r defined by HMAC-SHA-256(K, $PPR_{r,0}$)
RTP	Root page
RTX	Read transaction
$S_{d,s}$	512B sector s of LD_d
SLT	Global salt of type `Int[8]` randomly chosen by each instance of TreeTank
SN	Session bound to instance of TreeTank
TS_r	Time stamp of R_r
WTX	Write transaction

Table 3.1: TreeTank *acronyms. Italic font is used for acronyms and Courier font is used for types*

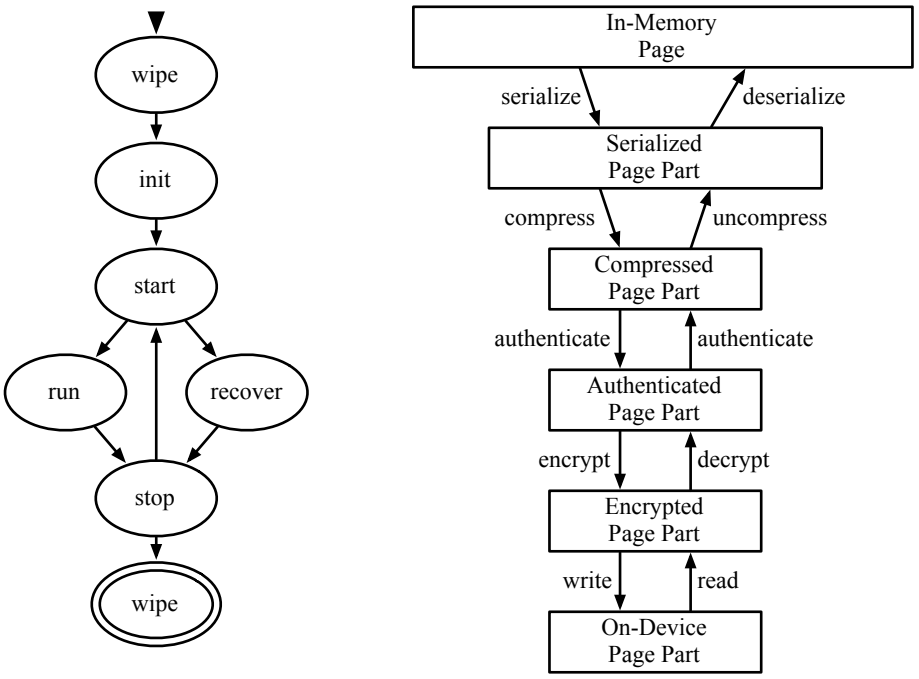


Figure 3.2: *The left figure displays the global state diagram. The right figure displays the process of (de-)serializing an in-memory page to or from persistent storage*

Initially, R_r inherits all logical pages from R_{r-1}. Subsequent modifications are only applied to copies of the logical pages and not to the originals. The copies are only visible to R_r. During a commit, the logical page tree is traversed in post order and all modified logical pages are sequentially stored as physical fragments ($F_{r,x}$, ..., $F_{r,0}$). If the whole logical page is stored, the physical fragment is named a page snapshot $PS_{r,p}$. If only a delta, i.e., a modification, to the last revision is stored, the physical fragment is named page delta $PD_{r,p}$. The logical and physical evolution of the revisions are depicted exemplarily in Figure 3.3.

A logical page $P_{r,p}$ is not serialized. $P_{r,p}$ consists of the logical page reference array $PRA_{r,p}$ and the node array $NDA_{r,p}$. $PRA_{r,p}$ contains all logical page references pointing to the logical child pages of $P_{r,p}$. $NDA_{r,p}$, contains all nodes of $P_{r,p}$.

The size of each array is fixed and defined for each page according to p. If the size is fixed to, say, 1024, the array offsets must be in [-1024, 1024]. The actual offset in the array is given by the absolute of the offset. A negative offset indicates that the logical page reference or node was modified before R_r. A positive offset indicates a modification during R_r. Offset 0 is reserved.

3.1.3 Confidentiality and Integrity

TreeTank can be run with either security turned off or on. If the security is turned off, *SLT* is set to 0 and the size of all tokens is 0. If the security is turned on, *SLT* consists of a random value not equal to 0.

As there can never be complete security against any threat, TreeTank chooses to protect against attacks according to Pareto's Principle – the 80/20 Rule – with simple, time-proven and efficient measures. The measures applied with TreeTank

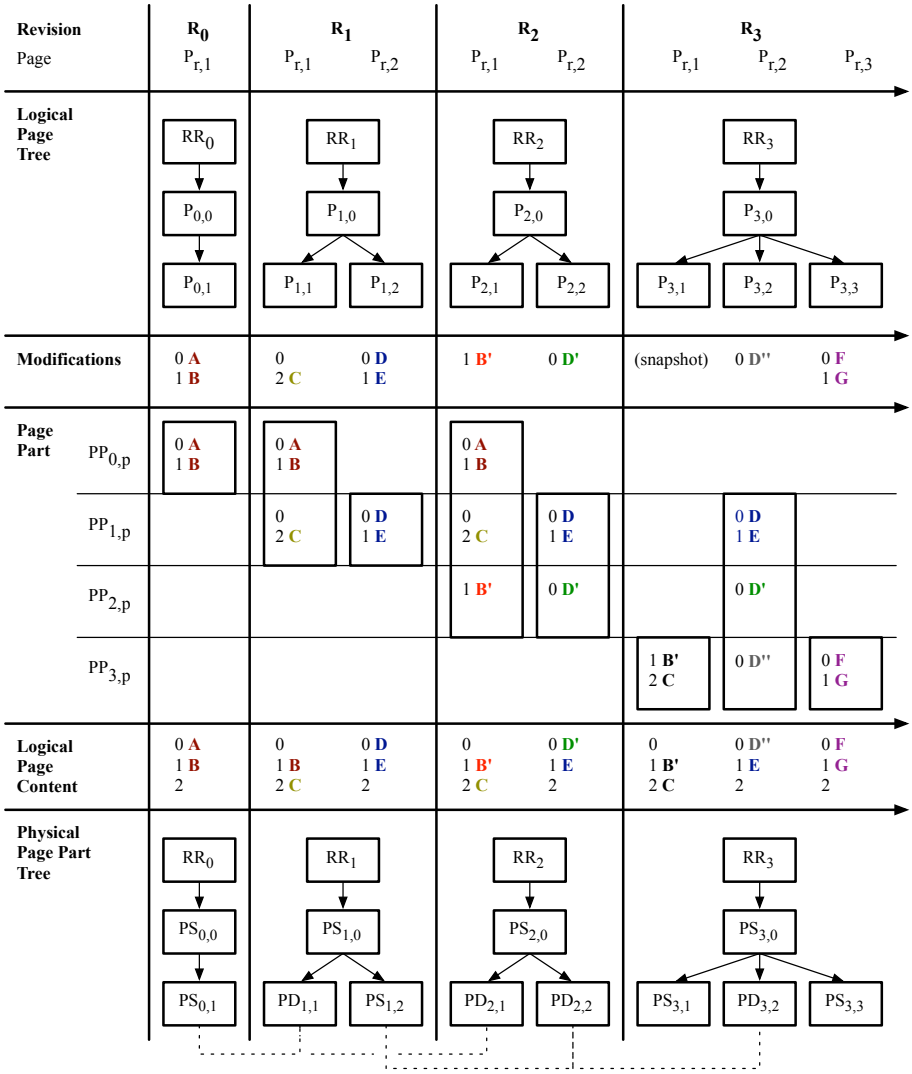


Figure 3.3: *Note that only node modifications are displayed while page references are ignored for simplicity of the description. The example shows node modifications applied within four revisions to three different pages. Each page stores at most three nodes and a full page snapshot is made in the fourth revision for the first page. The logic is as follows. The modification in revision R_0 inserts **A** as node 0 of page $P_{0,1}$ and **B** as node 1. This modification is physically stored in page snapshot $PS_{0,1}$ (note that this is equal to the page part $PP_{0,1}$ and that a page snapshot is created because it is the first appearance of this page part). Then, in R_1, node 0 of $P_{1,1}$ is deleted again and **C** is inserted as node 2. This modification is stored in page delta $PD_{1,1}$ (note that here, a page delta is used because there already existed a page snapshot in a previous revision). From a logical perspective, $P_{1,1}$ now contains two nodes, i.e., node 1 with value **B** and node 2 with value **C**. The logical perspective – with respect to the state – is constructed by reading two page parts in post order, i.e., $PD_{1,1}$ and then $PS_{0,1}$, populating the nodes only during their first appearance in a page part. As such, node 0 in $PD_{1,1}$ takes precedence over node 0 in $PS_{0,1}$. The logical perspective – with respect to the modifications – is reconstructed in pre order to reflect the sequence of modifications which can simply be derived from the node value. Then, in R_2, node 1 of $P_{2,1}$ is updated to the new value **B'**. This modification is stored in page delta $PD_{2,1}$. Finally, in R_3, no nodes are inserted, updated, or deleted in $P_{3,1}$, but a full page snapshot is taken. This snapshot is nothing but storing the logical page state in a new page snapshot $PS_{3,1}$. All other modifications on other pages follow the same logic as described for page $P_{r,1}$. If a page snapshot is taken in every fourth revision, at least one page part and at most three page parts must be retrieved to reconstruct the state or modifications within these four revisions*

force the attacker to get physical access to the computer and devices, use advanced know-how, and expensive equipment. Given that the master key is of high entropy and that cryptographic-strength randomness is used, it is currently thought to be impossible to learn anything from an offline device of ≤1PB storing an instance of TreeTank except that it is an instance of TreeTank.

Each instance of TreeTank is associated with a symmetric master key MK and a salt SLT, both of type `Int[8]`. MK and SLT are randomly chosen by TreeTank. MK must be kept private and is stored with the encrypted configuration in the header. SLT is made public and stored in the unencrypted part of the header. TreeTank uses MK and SLT to derive symmetric key K of type `Int[8]` through stretching and salting. $K = x_r$ where $x_0 = 0$, $x_i = \text{SHA-256}(x_{i-1} \parallel MK \parallel SLT)$, and $r \geq 2^{16}$.

The user provides the symmetric user key UK to encrypt and decrypt the configuration in the header. Again, a temporary key TK is derived from UK through salting and stretching. All other parts of TreeTank are encrypted with K. The separation of UK and MK allows the user to change the user key at any time without having to re-encrypt everything. For simplicity of description, it is assumed that the user generates and types MK. Though, it is highly recommended to generate MK by digitally signing SLT on a hardware token.

K is used to encrypt and decrypt H, RR_r, and $F_{r,p}$ with CTR-AES-256 [Fed01, NIS01a] operating with a block size of 16B. K is also used to authenticate H, RR_r, and $F_{r,p}$ with HMAC-SHA-256 [Fed02b, Fed02a]. CTR-AES-256 requires a unique nonce N of type `Int[2]` and a unique counter CTR of type `Long` for each encryption operation. Note that encryption and decryption with CTR-AES-256 are equivalent.

For the encryption of $F_{r,p}$, N is set to the first two `Int` of $FA_{r,p}$, and CTR is set to $FO_{r,p}$. CTR is then incremented for each 16B block of $F_{r,p}$ by one. For the encryption of H and RR_r, N is set to the first two `Int` of HMAC-SHA-256(K, SLT). If LD_1 and LD_2 are combined into LD_0, CTR is set to the device offset of H or RR_r in words. CTR is then incremented for each 16B block of H or RR_r by one. If LD_1 and LD_2 are not combined into LD_0, CTR is set to the negated device offset for all H or RR_r stored on LD_2 and decremented by one per 16B block.

3.1.4 Global State

The state diagram is depicted in Figure 3.2 and consists of the following states.

Wipe During the wipe state, all sectors are overwritten once or multiple times with a cryptographic-strength wipe algorithm [NIS01b].

Init The init state of TreeTank is as follows:

1. LD_1, LD_2, and MK are chosen by the user.
2. SLT is chosen by TreeTank.
3. K is derived from MK and SLT.
4. HT is set to HMAC-SHA-256(K, CNF).
5. H is set to (SLT ∥ CTR-AES-256(K, CNF ∥ HT)).
6. H is synchronously written to $S_{1,0}$, $S_{1,1}$, $S_{2,0}$, and $S_{2,1}$, and verified.

One verification failure causes a restart of the init state.

Start The start state of TreeTank is as follows:

1. LD_1, LD_2, and MK are provided by the user.
2. H is read from $S_{1,0}$, $S_{1,1}$, $S_{2,0}$, and $S_{2,1}$.
3. SLT is verified to be equal on all four replicas of H.
4. K is derived from MK and SLT.
5. All four replicas of H are decrypted to verify HT = HMAC-SHA-256(K, CNF).
6. $RR_{max(r)-1}$ is searched on LD_2 (or LD_1 if LD_2 is not available).
7. $RR_{max(r)-1}$ is verified to match $RR_{max(r)-1}$ on LD_1 (if LD_2 is available).

One verification failure causes a transition to the recover state.

Recover The recover state of TreeTank is as follows:

1. H is recovered.
2. LD_1 is recovered.
3. LD_2 is recovered.
4. After recovery, TreeTank goes into stop state.

Run The run state of TreeTank is as follows whenever a revision is committed:

1. $R_{max(r)}$ is synchronously written to LD_1.
2. $RR_{max(r)}$ is verified on LD_1.
3. $RR_{max(r)}$ is synchronously written to LD_2.
4. $RR_{max(r)}$ is verified on LD_2.
5. *max(r)* is set to *max(r)+1*.

One verification failure causes a 2nd serialization restarting with the first item.

Stop The stop state of TreeTank is as follows. New modifications are no longer allowed. Unwritten modifications are written as described with the run state. If the last item is not reached within a configurable timeframe, the TreeTank immediately stops. This prepares TreeTank to be safely started again or wiped.

3.1.5 On-Device Layout

The logical device layout is depicted in Figure 3.4. For simplicity of the description, all parameters and algorithms are explicitly chosen but can be adapted to individual requirements. A word is 32b wide and corresponds to `Int`.

TreeTank assumes the availability of at least one logical device LD. LD consists of an array of max(s) sectors ($S_{LD,0}$, ..., $S_{LD,max(s)-1}$), of 512B each. LD must support random read and write access to any $S_{LD,i}$. LD should be optimized for random reads and sequential writes. LD may grow repeatedly at any time by appending s' sectors ($S_{LD,max(s)}$, ..., $S_{LD,max(s)+s'-1}$). LD may limit the number of writes to any $S_{LD,i}$. TreeTank is optimized to reduce hot-spots, i.e., some $S_{LD,i}$ which is written to frequently and thus may age faster. Typically, flash-based devices are used as the underlying physical device. A logical volume manager provides the ability to dynamically grow LD.

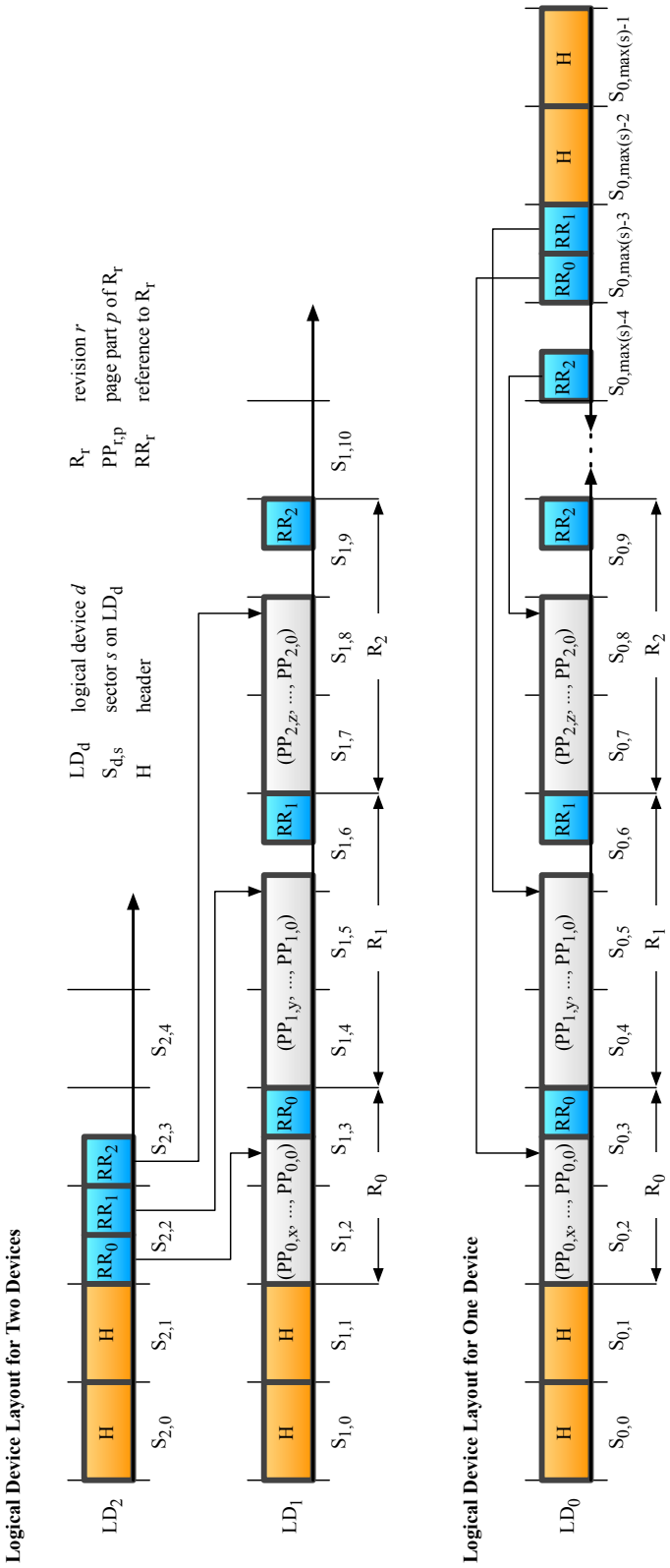


Figure 3.4: *Logical device layout for three revisions and the two alternatives, the top one to use two logical devices, the bottom one to use only a single logical device. Note that the two-device alternative has the advantage that it is not limited because new sectors can be added indefinitely to the right and that the single-device alternative is limited because it grows from the start and the end of the device*

TreeTank stores all data and metadata on the primary logical device LD_1. It should store the metadata on a secondary logical device LD_2 to speed up the operation of TreeTank. At any time, LD_2 can be reconstructed from LD_1. TreeTank may store incremental backups of LD_1 or LD_2 synchronously or asynchronously on one or more other LD. If TreeTank is used with a single LD_0 that must not grow, LD_1 and LD_2 can be combined into LD_0 by storing LD_1 starting at $S_{0,0}$ growing towards $S_{0,max(s)-1}$ and storing LD_2 starting at $S_{0,max(s)-1}$ growing towards $S_{0,0}$. While the sectors of LD_2 are reversed in order, their content is not.

The layout of LD_1 is as follows. $S_{1,0}$ holds the header H. $S_{1,1}$ holds a replica of the header H. The following sectors hold max(r) revisions (R_0, ..., $R_{max(r)-1}$). Each revision is of variable length and allocates a natural number of sectors. An R_r consists of three parts. First, a sequence of x fragments ($F_{r,x-1}$, ..., $F_{r,0}$), each $F_{r,p}$ of variable size and starting at a word-aligned offset. Second, a padding of zeroes. Third, a revision reference RR_r of 128B. RR_r is right-aligned within the last sector of R_r.

The layout of LD_2 is as follows. $S_{2,0}$ holds a replica of the header H. $S_{2,1}$ holds a replica of the header H. The following sectors hold an array of (RR_0, ..., $RR_{max(r)-1}$).

3.1.6 Basic, Complex, and XML Types

The serialization of all basic types either is of fixed or variable size and shown in Table 3.2. The value of a de-serialized basic type is equal for both the fixed-size and the variable-size serialization. Complex types are built from a combination of the basic types. The serialization of all complex types is shown in Figure 3.5. The complex types are described below.

Type	Size [B]	Description
`Byte`	1	Fixed-size byte in [-128, 127]
`Int`	4	Fixed-size integer in [-2^{31}, $2^{31}-1$]
`Long`	8	Fixed-size long in [-2^{63}, $2^{63}-1$]
`VarInt`	1 ... 5	Variable-size integer in [-2^{31}, $2^{31}-1$]
`VarLong`	1 ... 9	Variable-size long in [-2^{63}, $2^{63}-1$]
`Utf`	1 ... 4	Variable-size UTF-8 character
`T[t]`	$t \times$ size(T)	List of t items of type `T`

Table 3.2: *Serialization format of basic types*

Header Stores global information (see Table 3.3). The header H of type `Header` consists of the salt SLT of type `Int[8]`, the configuration C of type `Byte[448]`, and the header authentication HA of type `Int[8]`. C is system-dependent and used to tune the parameters of TreeTank. HA = HMAC-SHA-256(K, $SLT \parallel C$) is used to authenticate H. SLT, i.e., the first 32B of H, are not encrypted. H is stored twice on LD_1 in $S_{1,0}$ and $S_{1,1}$ and twice on LD_2 in $S_{2,0}$ and $S_{2,1}$.

Revision Reference Points to a revision (see Table 3.4). The revision reference RR_r of type `RevisionReference` consists of the fragment offset $FO_{r,0}$ of type `Long`, the fragment length $FL_{r,0}$ of type `Int`, the fragment authentication $FA_{r,0}$ of type `Int[8]`, the revision number of type `Long` the revision time stamp $RRTS_r$ of type `Byte[28]`, the reserved area of type `Byte[16]`, and the revision authentication RRA_r of type `Int[8]`.

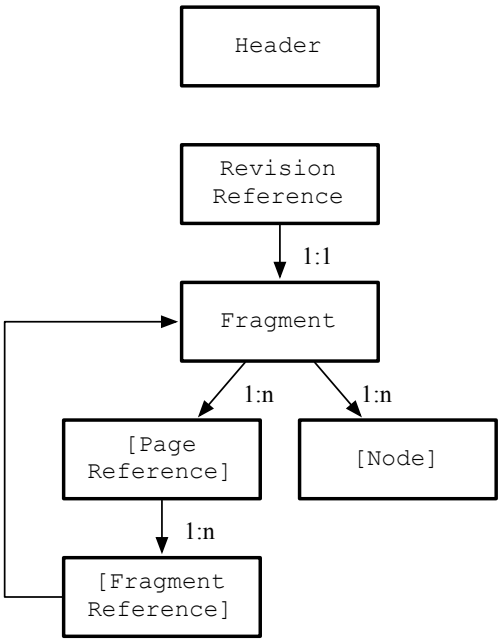


Figure 3.5: *Dependencies between serialized types*

Type	Description
`Int[8]`	Salt
`Byte[448]`	Configuration information
`Int[8]`	Header authentication

Table 3.3: *Fixed-size serialization format of header H of type* `Header`

Type	Description
`Long`	Fragment offset
`Int`	Fragment length
`Int[8]`	Fragment authentication
`Long`	Revision number
`Byte[28]`	Revision time stamp
`Byte[16]`	Reserved
`Int[8]`	Revision authentication

Table 3.4: *Fixed-size serialization format of revision reference* RR_r *of type* `RevisionReference`

RR_r points to $F_{r,0}$. RRA_r = HMAC-SHA-256(K, $FO_{r,0}$ ‖ $FL_{r,0}$ ‖ $FA_{r,0}$ ‖ $RRTS_r$ ‖ $RRNC_r$ ‖ `Byte[16]`) is used to authenticate RR_r. $FA_{r,0}$ = HMAC-SHA-256(K, $F_{r,0}$) is used to authenticate $F_{r,0}$.

For simplicity of description, it is assumed that RR_r has the same size as a sector. $RR_{max(r)-1}$ on LD_2 is found as follows. A binary search is performed on ($S_{2,2}$, …, $S_{2,max(s)-1}$), i.e., the array of revision references (RR_0, …, $RR_{max(r)-1}$). Each time the median $S_{2,m}$ is chosen, it is assumed to contain RR_{m-2}. If RRA_{m-2} = HMAC-SHA-256(K, $FO_{m-2,0}$ ‖ $FL_{m-2,0}$ ‖ $FA_{m-2,0}$ ‖ $RRTS_{m-2}$ ‖ $RRNC_{m-2}$ ‖ `Byte[16]`), the search is continued to the right, else to the left. If there is some $RR_{max(r)-1}$, the binary search will eventually select it.

The following algorithms apply if LD_2 is not available. Again, a binary search is performed with LD_1 as described above. If the median does not reveal a revision reference, the search is continued to the right and to the left as follows. Starting with $k_0 = 1$, the closest possible revision reference is searched in ($S_{1,m+k_{j-1}}$, …, $S_{1,m+k_j}$). If it is not found, k_j = -2 × k_{j-1}, until k_j reaches a configurable boundary.

Root Node Stores revision-wide information (see Table 3.5). The root node $RN_{r,0,1}$ of type `RootNode` consists of an inherited node of type `Node`, the author size `$1` of type `VarInt`, the author of type `Utf[$1]`, the comment size `$2` of type `VarInt`, the comment of type `Utf[$2]`. $RN_{r,0,1}$ must be located in $P_{r,0}$ at offset 1, i.e., it is updated with each new revision. $RN_{r,0,1}$ stores information collected during a write transaction commit.

Type	Description
`Node`	Inherited node
`VarInt`	Author size `$1`
`Utf[$1]`	Author
`VarInt`	Comment size `$2`
`Utf[$2]`	Comment

Table 3.5: *Variable-size serialization format of root node $RN_{r,p,o}$ of type `RootNode`*

Page Reference Intermediate references to other pages (see Table 3.6). The page reference $PR_{r,p}$ of type `PageReference` consists of the page reference array offset of type `VarInt`, the fragment reference count `$1` of type `VarInt`, and the list of fragment references of type `FragmentReference[$1]`.

RR_r references $F_{r,0}$. All other pages are referenced through a page reference $PR_{r,p}$ either referencing a single $F_{r',p}$ where $r' = r$, or a sequence of $F_{r'',p}$, …, $F_{r^*,p}$ applied to an initial $F_{r',p}$ to retrieve $P_{r,p}$ where $r' < r'' < \ldots < r^* < r$. $FC_{r,p}$ denotes the number of fragments required to fully reconstruct $P_{r,p}$.

Type	Description
`VarInt`	Page reference array offset
`VarInt`	Fragment reference count `$1`
`FragmentReference[$1]`	Fragment reference list

Table 3.6: *Variable-size serialization format of page reference $PR_{r,p}$ of type `PageReference`*

Fragment Reference Set of page references or nodes (see Table 3.7). The fragment reference $FR_{r,p}$ of type `FragmentReference` consists of a fragment revision of type `VarLong`, a fragment offset of type `VarLong`, a fragment length of type `VarInt`, and the fragment authentication $FA_{r,p}$ of type `Int[8]`. $FA_{r,p}$ = HMAC-SHA-256(K, $F_{r,p}$) is used to authenticate $F_{r,p}$.

Type	Description
`VarLong`	Fragment revision
`VarLong`	Fragment offset
`VarInt`	Fragment length
`Int[8]`	Fragment authentication

Table 3.7: *Variable-size serialization format of fragment reference $FR_{r,p}$ of type `FragmentReference`*

Fragment Set of fragment references (see Table 3.8). The fragment $F_{r,p}$ of type `Fragment` consists of a page reference count `$1` of type `VarInt`, a page reference list of type `PageReference[$1]`, a node count `$2` of type `VarInt`, and a node list of type `Node[$2]`. The page reference list stores modifications to any page reference pointing to the child pages of $P_{r,p}$. The node list stores modifications to any node of $P_{r,p}$.

Type	Description
`VarInt`	Page reference count `$1`
`PageReference[$1]`	Page reference list
`VarInt`	Node count `$2`
`Node[$2]`	Node list

Table 3.8: *Variable-size serialization format of fragment $F_{r,p}$ of type `Fragment`*

Node Stores the actual payload (see Table 3.9). The node $ND_{r,p,o}$ of type `Node` consists of a node array offset of type `VarInt`, and a node type of type `VarInt`. Node type -1 indicates a deleted node. Node type 0 is reserved for the root node type. The actual node type is given by the absolute of the node type. All other nodes inherit these two fields.

The XML node types can be serialized according to Table 3.10. Note that this list can easily be extended by other XML node types such as comment or processing instruction, or also with non XML node types for non-XML scenarios.

Type	Description
`VarInt`	Node array offset
`VarInt`	Node type

Table 3.9: *Variable-size serialization format of node $ND_{r,p,o}$ of type `Node`.*

(a) Node Types

Value	Type
0	`RootNode`
1	`XMLDocumentNode`
2	`XMLElementNode`
3	`XMLTextNode`
4	`XMLAttributeNode`
5	`XMLNamespaceNode`

(b) XMLDocumentNode

Type	Description
`Node`	Inherited node
`VarLong`	First child key
`VarLong`	Child count

(c) XMLElementNode with inlined attributes

Type	Description
`Node`	Inherited node
`VarLong`	Parent key
`VarLong`	First child key
`VarLong`	Left sibling key
`VarLong`	Right sibling key
`VarLong`	Child count
`VarInt`	Attribute count `$1`
`Attribute[$1]`	Array of attributes
`VarInt`	Namespace declaration count `$2`
`NamespaceDeclaration[$2]`	Array of namespace declarations
`VarInt`	Name key
`VarInt`	URI key

(d) XMLTextNode

Type	Description
`Node`	Inherited node
`VarLong`	Parent key
`VarLong`	Left sibling key
`VarLong`	Right sibling key
`VarInt`	Value type
`ValueType`	Value

(e) XMLAttributeNode

Type	Description
`VarInt`	Name key
`VarInt`	URI key
`VarInt`	Value type
`ValueType`	Value

(f) XMLNamespaceNode

Type	Description
`VarInt`	URI key
`VarInt`	Prefix key

Table 3.10: *Node types. Note that additional types such as processing instruction can be added later on demand*

3.2 SlidingSnapshot

Log-structured (copy-on-write) storage systems write modified pages to free space, i.e., they do not overwrite existing pages. In addition, versioning systems store pages, which are only partially modified, as incremental or differential deltas to reduce the overall storage consumption. To reconstruct the full page, they intermittently store full page snapshots and calculate the page by applying all deltas to this snapshot. Snapshots are triggered by the number or the size of the intermediate deltas. This is not optimal for three reasons. First, a variable number of deltas must be be applied in a CPU-intensive procedure to one intermediate snapshot – a task which takes a hardly predictable amount of time. Second, frequent modifications between alternating parts of the page still consume too much storage space. Third, the size of the written data greatly varies because of the small deltas compared to the large full snapshots.

SlidingSnapshot is a storage technique to reduce the required storage space but in the worst case is not worse than the original approach. In addition, the size of the written data is more evenly distributed because the snapshot is distributed throughout the deltas (sliding). Finally, this technique allows to retrieve a page with predictable and significantly less time and CPU requirements.

Therefore, a page is divided into cells as depicted in Figure 3.6. Each cell is uniquely identified by its cell number within its page. Each page modification can hit one or more cells and a modification can either be an insert, an update, or a deletion. Every page modification creates a new version of the page and physically results in a page delta, which contains the following. First, a new absolute version number. Second, the cells which were modified during this version (including the type of modification). Third, a snapshot of the cells which have not been modified since the past N versions of the page. Each cell snapshot contains the difference between the new version and the version number when the cell snapshot was last modified.

In each version, one can access the page with a pointer. This pointer contains N pointers to access the page delta. These N page deltas can be fetched in parallel. Typically, N is constant and less than 10. However, N can be varied per page according to the encountered workload and if the predictability is not so important.

For N equal to 2, each cell of a page is stored at most twice in at most two deltas: At least as the latest version of this cell and at most as the latest and second-latest version of this cell. If the second-latest version of the cell is not in one of the deltas, it can be accessed by reading the page version defined by the version difference stored in the cell.

SlidingSnapshot cleverly balances the full, incremental, and differential snapshots with a sliding snapshot consisting of exactly N deltas. This has the following advantages and is extensively evaluated in [Gra14]. First, the space-efficiency. In the best case, it only uses c/m th of the space of the full page snapshot (c is the number of changed nodes, m the total number of nodes on the page). On average, it uses less than, and in the worst case, it uses as much space as the full snapshot. Second, the balanced I/O. Read and write sizes do not wary as much as with the full snapshot because every I/O operation contains the sliding part of a snapshot alongside with the actual data. Third, the predictability. We guarantee the retrieval of any state or sequence of modifications for any page to take place within a predefined amount of time, I/O, and CPU effort, thus providing real-time capabilities.

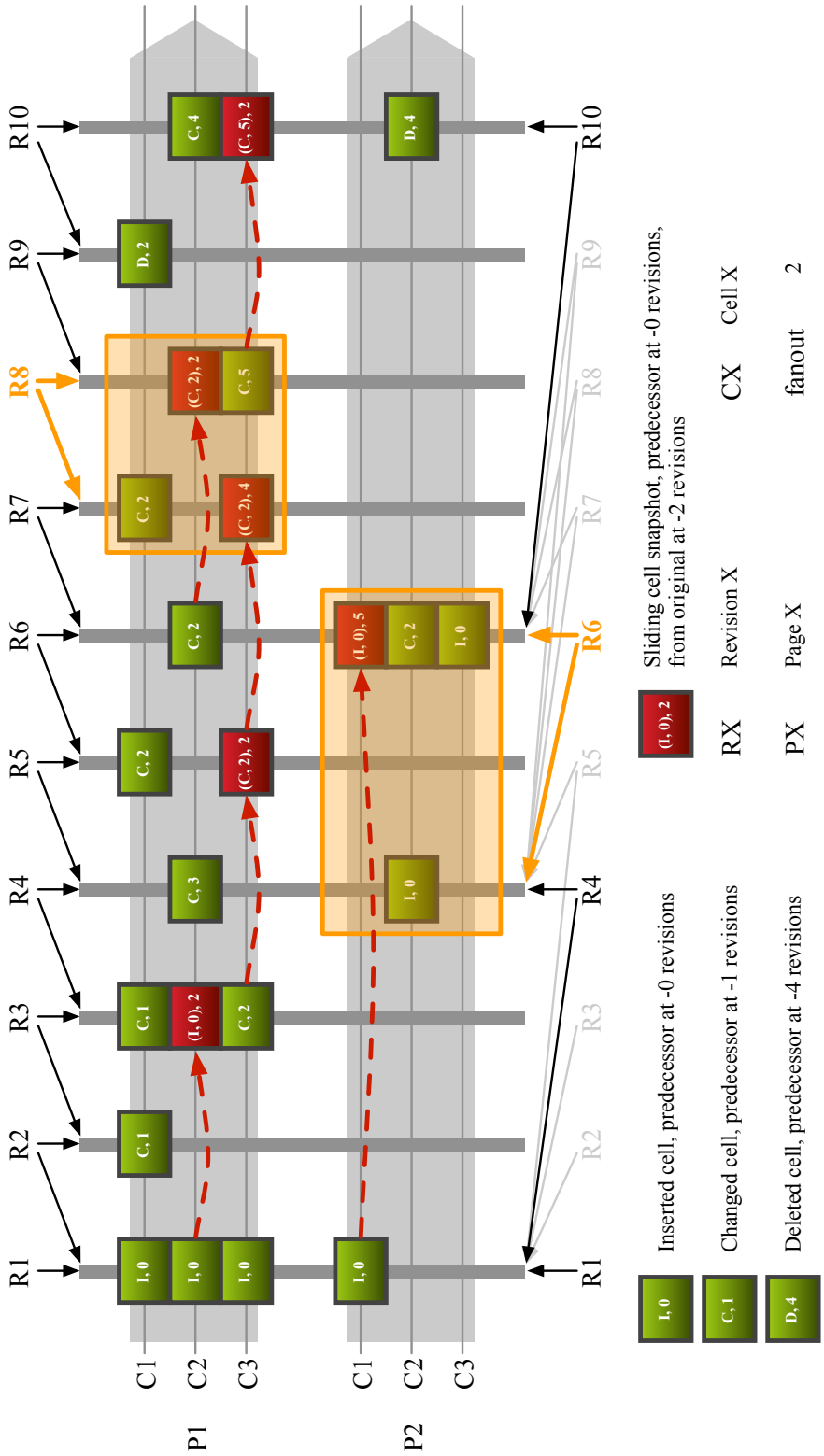


Figure 3.6: *This figure illustrates* SlidingSnapshot *for two example node pages evolved throughout ten versions. Node page P1 is frequently modified and node page P2 is rarely modified. The fanout is set to two, i.e., at most two node pages must be read to access any node at any version. The orange boxes denote the amount of data to be read to reconstruct state or modifications of a given page in a given revision*

3.3 Distribution

In contrast to relational databases the distribution of document-centric XML is not well researched. While there are some suggestions on how to split and distribute large XML documents, these approaches do not consider the parallel query evaluation. In this section, we summarize five different algorithms based on the work of Sebastian Graf to search after suitable split nodes in a large XML document [GKW08]. While this is just an excerpt, we feel that the distribution aspect is very important to further improve the performance and availability of our *evolutionary* tree-structured storage. It describes how to distribute extractable sub-structures over a fixed number of peers and how to query these peers in parallel to retrieve the final result.

Initially, XML was just used to exchange or store small amounts of data in an unified way. Nowadays, even large data- and document-centric data sets are stored in XML files or (native) XML databases. A single system storing such enormous amounts of XML data quickly comes to its limits due to insufficient amounts of storage space, processing power, or available main memory. When it comes to massive concurrent access or to single-system failures, distribution to multiple systems becomes the only feasible option.

The distribution of data-centric XML is straightforward and extensively researched with relational databases where the aim is to distribute columns and rows in a reasonable way. Here, reasonable means with minimal effort and optimally suited to the workload to which the database is exposed. In stark contrast, the distribution of document-centric XML is much more difficult. The straightforward approaches for distributing XML fragments will likely not result in a balanced system. E.g., if a splitting algorithm splits the XML tree at the first level, i.e., at the root node, some XML fragments are likely to contain large sub-trees while others might only contain a few nodes.

It is a challenging task to automatically find the appropriate split nodes in the tree because the system has to adapt to every single XML document and therein to a potentially completely different topology for each sub-tree. In a system with a fixed number of peers, it is important to split the tree in a way to make sure that every single peer gets its fair share of XML fragments of comparable size. In other words, the question is whether there exists a split algorithm which produces an optimal distribution for any document-centric XML data. Only when the XML fragments are evenly distributed over multiple peers, each one has an equal chance of being involved in a parallel query evaluation. Still, chances are that some peers get more involved due to a specific query workload. If the distribution is already skewed due to a bad split algorithm, even simple queries will create hot-spots and therefore bottlenecks in the whole system.

During the quest for the optimal split algorithm, Sebastian Graf developed five different approaches and analyzed each one based on three well-known large XML data sets. The first is XMark [SWK+01], a generic data set, the second is Treebank [MMS93], a corpus of linguistic documents, and the third is DBLP, a listing of publications relevant to computer science [Ley02b]. All three data sets contain a mix of mainly document-centric but also data-centric aspects. For each data set, he evaluated two different XPath expression classes. The first expression class is a depth-first search consisting of a concatenation of many child axis steps, the second expression is a breadth-first node count for a descendant step.

Two independent steps are required to achieve a split: First, suitable split nodes must be identified. These nodes (and their respective sub-trees) are extracted from the original tree structure as described in [BC07]. Second, the sub-structures must be stored in a given number of peers to achieve an independence between the number of peers and the number of extractable sub-structures. For the split algorithm to work, the following items must be fulfilled:

- The split nodes should have equal tag names. If the tag names of the root-nodes of the desired sub-structures are all unique, it is not guaranteed that tree-walking queries working with node-tests can be processed in a parallel way due to the different semantic aspects of the root nodes of the extractable sub-structures.
- The identified sub-structures should have a maximum size. Working on a distribution in a client/server environment generates an overhead. The handling of this overhead can only be justified if the evaluation of the fragments is expensive. So the sub-structures should be as big as possible to ensure that the distribution leverages a performance benefit.
- The resulting fragments should have an equal number of nodes. Regarding the target of a parallel evaluation, the parallelism is used as long as each peer is processing its data. Therefore to ensure a long-term parallelism, every peer should have the same amount of data to process.
- The split algorithm should work without any meta-data. Every input to the split operation must be based on a before-hand exploration of the structure or at least of the DTD. Regarding document-centric data, an analysis with the focus on structural and semantic constraints, is not easy. If the split algorithm itself has knowledge about the XML instance, dependencies to third-party inputs are minimized.

Sebastian Graf suggests the following split algorithms [GKW08]:

LevelSplit The first approach marks all nodes on a given level as split nodes. This approach works perfectly for data-oriented XML where the structure is quite similar as described with current approaches based on horizontal fragmentation techniques inherited from the distribution of relational data. However, it is not guaranteed to get a good fragmentation result with this approach regarding document-centric data due to the irregularity of the structure. Therefore, the LevelSplit possibly results in undesirable sub-structures. An additional analysis must be performed to find the suitable levels in a given XML instance. Especially regarding document-centric data with highly irregular structures, the extracted sub-structures are potentially not matching the defined requirements.

FanoutSplit To get only large sub-structures, it is assumed that a node with a large sub-tree has a large fanout as well. Therefore, according to a given number of children, a node in the XML structure is marked as a split node if this number exceeds a given threshold. This threshold must be set in advance of the identification process. Consequently, a suitable exploration becomes inevitable. Again, this approach works well for data-centric XML. For document-centric structures, the identification of the split nodes should not be solely based on the fanout of a node due to the irregularity of the document structure, as this can result in nodes with a large fanout but relatively small

sub-trees. Additional to this possible violation of the design objectives, there is no assertion that there are no unique tag names in the set of split nodes.

SemanticSplit To consider equal tag names of split nodes, this approach is based on the occurrence of tag names. On each level, the occurrences of different tag names on the sibling axis are counted. If there is more than one node in the sibling axis, and each sibling has the same tag name, these nodes are identified as split nodes. Unfortunately, as with the LevelSplit, the identified sub-structures are quite small and therefore not optimal. The partitioning of sub-structures makes more sense with respect to parallel evaluations. Moreover, an exploration to find suitable nodes is unnecessary. Additionally the extracted sub-structures have a high similarity due to of the tag names of their root node. This assures a parallel evaluation of these structures even if the sub-structures themselves are small.

PostorderSplit To tackle the need for large extractable sub-structures, a split algorithm based on the designated number of fragments was developed. With the help of a post order traversal through the original tree structure, the sub-tree size corresponding to each node is computed. A threshold $\frac{n}{i*2}$ based on the nodes n in the XML instance and the number of available peers i is computed. At each node, the number of processed nodes is compared to the threshold. If the threshold is attained, the actual node is marked as a split node and the counter of nodes is reset. If no node has the potential to work as a split node, the children on the first level are selected to achieve at least a basic fragmentation. The split node obviously generates sub-structures with a similar load. Regarding data-centric XML, this approach works well as the identified split nodes have equal tag names.

PostorderSemSplit To get rid of the possibly different tag names in the extracted sub-trees based on the PostorderSplit, the PostorderSemSplit combines the post order traversal with the SemanticSplit. The tree structure is processed with a post order traversal similar to the PostorderSplit. However, instead of computing the size of the current sub-tree according to a given node, the sizes of the processed nodes according to the tag name of the current node are stored. This size must attain the same threshold as described in the PostorderSplit. If the threshold is attained, all nodes with the given tag name are marked as split nodes. With this approach, the main objectives are satisfied. The extracted sub-trees have an adequate large size, except too highly recursive occurrences of the same tag name. In this case, this approach can lead to small sub-structures. The similarity of the root-nodes of the extractable sub-structures is ensured by the computation of traversed nodes according to the tag names. As the identification of the sub-structures is based on the number of nodes and the number of available peers, this split-operator is not in need of a previous exploration of the XML instance. However, if the XML data is based on a recursive DTD this approach could result in suboptimal splits, because the number of corresponding nodes related to a tag name can be falsified by the recursive occurrence of the elements.

After the identification of suitable split nodes, with focus on a parallel evaluation of the corresponding sub-structures, these sub-structures have to be extracted and stored in different peers. The distribution of the identified structures is conducted by a simple *round-robin* algorithm. A suitable number of fragments according to the number of available peers is chosen. Additional to these structures, a root fragment is initialized. Afterwards, the original XML dataset is traversed in pre order. Each

node is inserted in the root fragment until the first split node is reached. Then, a proxy-node with an unique ID is inserted into the root fragment and a child fragment is selected in *round-robin* order. On the child fragment, a corresponding proxy node is inserted which also has an unique ID. With these IDs, a direct access between the proxy node in the root fragment and the proxy node in the child fragment is obtained. After inserting this proxy node, the split node itself is inserted beneath this proxy node. Subsequently, each following node is inserted on this child fragment until the traversal is leaving the current sub-tree. Finally, the following nodes are inserted into the root fragment until the next split node is reached and so on. Even document-centric data, depending on suitable split nodes, is very well fragmented with this approach because split nodes with the same tag name are distributed in *round-robin* fashion over all peers.

These findings affirm the assumption that a trivial split algorithm does not consistently achieve an optimal distribution. The summarized automatic split algorithms, though, can split large XML documents with the same overhead as a trivial split algorithm, but with much better scalability when it comes to parallel query evaluation.

The similarity of the split nodes and the equality of the load can be positive as well as negative regarding the LevelSplit and the FanoutSplit. This depends on the selected level or the chosen threshold. The SemanticSplit scales well for data-centric data. Regarding document-centric data, a good fragmentation can not be guaranteed. The fragments of the PostorderSplit are as equal as possible regarding the structure of the original XML data. This equality is available with the PostorderSemSplit as well in most cases. Only with highly recursive structures, multiple small sub-structures are identified. Yet, in all of the test cases, no negative fragmentation was achieved with the PostorderSemSplit and a high number of participating peers (see [GW08] for further information).

We see many open issues in the area of distributing large-scale XML data for parallel query evaluation:

- Depending on the query, a simple parallel evaluation may need to reorder the result retrieved from multiple peers. We want to investigate which queries are affected and whether this reordering operation can be prevented or efficiently done during either the split operation or the parallel evaluation.
- While XPath provides a fundamental idea of what the evaluation time will be, it is only a sub-set of current query languages such as XQuery. We want to look at XQuery and how it can be executed in parallel by rewriting the query itself or by optimizing and splitting the logical operator tree.
- Currently, we split a static XML document. We are interested in updates and how they lead to re-assignments of the XML fragments to keep the whole distribution in balance. This may be achieved through moving the fragments themselves or by dynamically further splitting up the XML fragments.
- The availability of indices may lead to faster evaluations for certain queries, i.e., an index can usually answer the query in logarithmic time without the need to do a full XML fragment traversal. However, an index will incur more update overhead and may itself grow so large that it also must be distributed. E.g., a full text index which has to store a term occurring in a large percentage of the nodes is no longer useful. Through splitting the XML document in smaller parts, we also make sure to split the domain of each index and potentially reduce the over-all update and search time.

- The reliability and availability of large XML documents will also become an issue. E.g., it is no longer possible to export Wikipedia to an XML file within a single day. Loosing the whole file due to a peer failure is catastrophic and would interrupt a service relying on it for too long. As soon as Wikipedia is distributed, the loss of a single peer will only erase a small part of the overall document. The question then becomes how to store a single XML fragment on multiple peers and how to exploit this knowledge to further speed up the query evaluation when each peer has its individual performance and variable network connection quality.

3.4 Summary

The unified storage manager combines many, often separately implemented features, into one concept. Most importantly, it shows, that these concepts can be applied at the much finer-granular node-level and are not limited to the file-level. This opens a wealth of opportunities, as, e.g., the proposed declarative access to file system data beyond the file-level barrier by Alexander Holupirek [Hol06, Hol12]. Essentially, it paves the was for a paradigm shift towards the *evolutionary* approach.

The general-purpose concept of SlidingSnapshot shows, that secure node-level copy-on-write can be combined with space-efficient, predictable and realtime node-level access. Space-efficiency is an important property when keeping the full modification history of a node. SlidingSnapshot uses at most as much space as traditional file- or page-level algorithms but allows for many workloads where it uses less space. In the optimal case, it only uses c/m *th* of the space of the full page snapshot (c is the number of changed nodes, m the total number of nodes on the page). Most importantly, SlidingSnapshot is not limited to our TreeTank concept but can be integrated with any versioning or backup system as well as other data structures than trees.

We summarized the aspect of distribution with regard to performance and availability. It is shown that good split algorithms are key to the distribution and efficient querying of large tree-structured data such as XML. However, much work remains before a cluster of peers will automatically and collaboratively store and query such large-scale XML data sets.

Chapter 4

Evaluation

We spent a lot of work on the background and concepts of TreeTank and SlidingSnapshot. We felt that it was important to verify and analyze our ideas based on tangible implementations. In fact, we implemented two generations of TreeTank. First, a version to affirm its linear scalability (Section 4.1). Second, a version to show that node-level granularity is feasible without giving up the linear scalability (Section 4.2). The implementations also proved invaluable for our work on interfaces and applications. In addition, it evolved three tools, one to provide block device access from high-level languages, one to create solid benchmark results, and one to monitor and visualize block access (Section 4.3).

It is important to outline that linear scalability is a major design goal. If the application scales with more data simply by adding new resources at a fixed cost without compromising on performance or redesigning the application, then the application has linear scalability.

4.1 Linear Scalability

Two competing encoding concepts are known to scale well with growing amounts of XML data: XPath Accelerator encoding implemented by MonetDB for in-memory documents and X-Hive's Persistent DOM for on-disk storage. We identified two ways to improve XPath Accelerator and present prototypes for the respective techniques: Christian Grün's BaseX boosts in-memory performance with optimized data and value index structures while TreeTank introduces native block-oriented persistence with logarithmic update behavior for true scalability, overcoming main-memory constraints.

The easy-to-use Java-based benchmarking framework PERFIDIX (see Subsection 4.3.2) was developed and used to consistently compare these competing techniques and perform scalability measurements. The established XMark benchmark was applied to all four systems under test. Additional full-text-sensitive queries against the well-known DBLP database complement the XMark results.

Not only did the latest version of X-Hive finally surprise with good scalability and performance numbers. Also, both BaseX and TreeTank hold their promise to push XPath Accelerator to its limits: BaseX efficiently exploits available main memory to speedup XML queries while TreeTank surpasses main-memory constraints and

rivals the on-disk leadership of X-Hive. The competition between XPath Accelerator and Persistent DOM definitely is relaunched.

XML has become the standard for universal exchange of textual data, but it is also increasingly used as a storage format for large volumes of data. One popular and successful approach to store and access XML data is to map document nodes to relational encodings. Based on the XPath Accelerator encoding [Gru02], the integration of the Staircase Join [GvKT03], an optimizing to-algebra translation [BMR05], and an underlying relational engine that is tuned to exploit vast main memories, the MonetDB system currently provides unrivalled benchmark performance for large XML documents [BGea06]. This section describes and evaluates two lines of research aiming at further improvements. We implemented two experimental prototypes, focusing on current weaknesses of the MonetDB system: the lack of value-based indexes and the somewhat less thrilling performance once the system gets I/O-bound. Separate prototypes have been chosen because they allow us to apply independent code optimizations.

Evaluation. The widely referenced XMark benchmark [SWK+02] and the well-known DBLP database [Ley02a] were chosen for evaluation to get an insight into both the current limitations and the proposed improvements of XPath Accelerator. BaseX is directly compared to MonetDB [Bon02] as both are main-memory based and operate on a similar relational encoding. TreeTank is compared to X-Hive [XH05b] as both are mainly based on secondary storage. The main difference to TreeTank is that X-Hive stores the XML as a Persistent DOM [HMF99]. Both MonetDB and X-Hive were chosen because current performance comparisons [BGea06] suggest them to be the best available solutions for either in-memory or persistent XML processing. In the course of this work, published results about MonetDB and X-Hive could be revisited. Both BaseX and TreeTank strive for outperforming their state-of-the-art competitors.

Contributions. Our main contribution is the evaluation of our approaches against the state-of-the-art competitors, using our own benchmarking framework PERFIDIX. In more detail: The first prototype, BaseX, pushes in-memory XML processing to its limits by applying thorough optimizations and an index-based approach for processing predicates and nested loops. The second prototype, TreeTank, linearly scales XPath Accelerator beyond current memory limitations by efficiently placing it on secondary storage and introducing logarithmic update time. Finally, the use of our benchmarking framework PERFIDIX assures a consistent and reproducible benchmarking environment.

Outline. The improvements to the XPath Accelerator encoding and its implementation are described by introducing our two prototypes. We intensively evaluated and compared the state-of-the-art competitors in the field of XML-aware databases against our optimized prototypes. In Subsection 4.1.6, we summarize our findings and have a look at our future work.

Attracted by the approved and generic XPath Accelerator concept, we were initially driven by two basic questions: how far can we optimize main-memory structures and algorithms to efficiently work with the relational encoding? Next, which data structures are suitable to persistently map the encoding on disk and allow continuous updates? Two different implementations are the result of our considerations: BaseX creates a compact main-memory representation of relationally mapped XML documents and offers a general value index, and TreeTank offers a sophisticated native persistent data structure for extending the XML storage to virtually unlimited sizes.

4.1.1 Optimized In-Memory Processing

BaseX was developed by Christian Grün to push pure main memory based XML querying to its limits both in terms of memory consumption and efficient query processing. Main memory is always limited, compared to the size offered by secondary storage media, so we aimed at optimizing the main-memory representation of the XML data to overcome the limitations. Main-memory XQuery processors such as Galax [FSC+03] or Saxon [Kay98] work efficiently on small XML files, but querying gets troublesome for larger files as the temporary XML representations occupy 6 to 8 times the size of the original file in main memory. MonetDB [Bon02] is designed as a highly efficient relational main memory database, and its Pathfinder/XQuery extension [BGea06] applies relational operations to process XML node sets at an amazing speed.

Our first Java prototype can be seen as a hybrid between relational and native XML processing. It uses a relational, edge-based *pre/parent* encoding for XML nodes, but issues arising from the set-oriented approach, such as the orderedness of XML node sets, can be evaded as all the data can be processed sequentially, thus simplifying orderedness for location step traversals. The current implementation can be used both as a real-time XML query tool and as a query application. XPath expressions can be passed on via the command line. Alternatively, an interactive mode is available to directly enter queries which are processed locally or by a BaseX server instance. XML files can be saved as database files to avoid future parsing of the original documents.

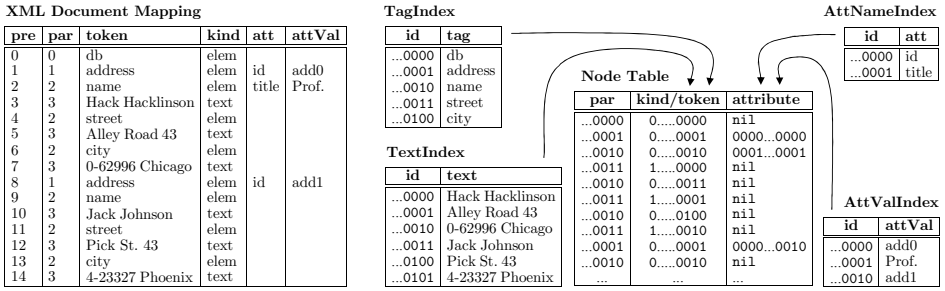


Figure 4.1: *Relational mapping of an XML document (left), internal table representation in BaseX (right)*

Data structures. The system is mainly built on two simple yet very effective data structures that guarantee a compact mapping of XML files: an XML node table and a slim hash index structure. The relational structure is represented in the node table, storing the *pre/parent* references of all XML nodes; the left table of Figure 4.1 displays a mapping for the XML snippet shown in Figure 4.2. The table further references the token of a node (which is the tag name or text content) and the node kind (element or text). Attribute names and values are stored in a two-dimensional array; a nil reference is assigned if no attributes are given. All textual tokens – tags, texts, attribute names and values – are uniformly stored in a hash structure and referenced by integers. To optimize CPU processing, the table data is exclusively encoded with integer values (see Figure 4.1, right tables).

To save memory, some table values store more than one attribute. Based on the extensive evaluation of numerous large XML instances up to 38GB, we noted that the maximum values for token references showed out to be constantly smaller than

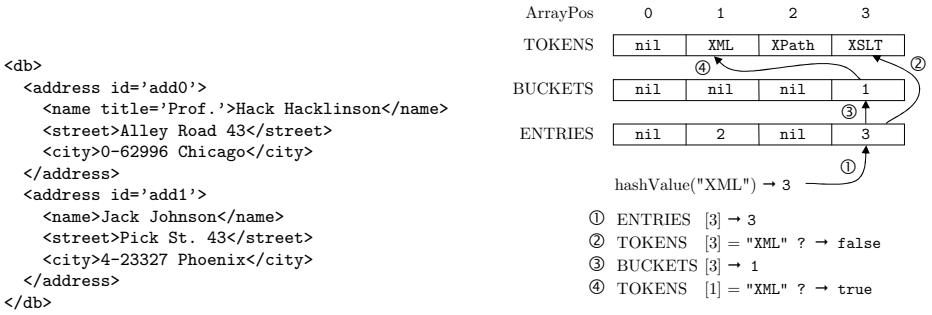


Figure 4.2: *The left side shows the XML snippet, mapped in Figure 4.1. The right side shows the hash index structure: finding 'XML'*

32bit, and as the node kind requires only 1bit, the two values are merged into one integer. The remaining space will be used for additional node information in future. The attribute name and value references are merged as well, sharing 10 and 22bits, respectively. The original values can be efficiently accessed via CPU-supported bit shifting operators.

The second data structure, the hash index, is an implementation of a linked list hash structure. Some optimization was done to minimize memory usage and to allow very quick access to the stored tokens. Similar to the table structure, the hash index was flattened to work on integer arrays. All arrays are sized by a power of two, and the bitwise AND operator (&) cuts down the calculated hash value to the array size. This approach works faster than conventional modulo (%) operations as the common CPU operators are directly addressed. Moreover the array size does not rely on the calculation of prime numbers, and the index structure can be quickly resized and rehashed during index generation, leading to a quick document shredding process.

Three arrays suffice to organize all hash information (see Figure 4.2). The TOKENS array references the indexed tokens. The ENTRIES array references the positions of the first tokens, and the BUCKET array maps the linked list to an offset lookup. After a hash value has been calculated for the input token and trimmed to the array size, the ENTRIES array returns the pointer to the TOKENS array. If the pointer is nil, no token is stored; otherwise the token is compared to the input. If the comparison fails, the BUCKET array points to the next TOKENS offset or yields nil if no more tokens with the same hash value exist.

Querying. The integrated XPath parser evaluates basic XPath queries, including all XPath axes, node tests and basic predicates (with textual, numeric and positional matches). Similar to MonetDB, BaseX is built on a step-based path execution. It has been pointed out that intermediate context sets can get pretty large whereas the result set is often small [BKS02]. This observation is particularly obvious when commonly used and highly selective predicates occur in the query, such as those pointing, e.g., to single attribute ids. To speedup predicate queries, we thus chose to introduce a general value index [MW99] for all text nodes and attribute values. The index is applied when exact string comparisons are found in the query. Moreover, it is also very effective when nested loops with predicate joins are encountered, reducing the quadratic to a linear execution time without the need of additional algorithms. More detailed information can be found in the evaluation in Subsection 4.1.5.

The value indexes are implemented pretty straightforward. The existing text and attribute value indexes are extended by references to the table's *pre* values, resulting in an inverted list. The correct use of the index is a little bit trickier: as the index returns a node set for the specified predicate, the XPath query has to be inverted and rewritten. Descendant steps are converted into ancestor steps and vice versa, and predicates are moved. In the following query doc("XMark.xml")/site//item[@id = "item0"], item0 is matched against the attribute value index. The resulting context set is matched against the item parent, the site ancestor and doc("XMark.xml") as the initial context node. The internal query reformulation thus yields index::node() [@id = "item0"]/parent::item[ancestor::site/parent::doc()]. A detailed analysis on rewriting reverse to forward axes can be found in [OMFB02].

The Staircase Join [GvKT03] offers three basic concepts to speed up path traversal: Pruning, Partitioning and Skipping. The basic ideas behind all three concepts could be modified and applied on the *pre/parent* encoding, thus accelerating step traversals by orders of magnitudes. Note that the Staircase Join demands the storage of an additional *post* or *size* value. This is why no exact equivalence can be derived for the exclusive *pre/parent* representation. Depending on the current node set, further speedups are applied. One of them is described in more detail: the Pruning algorithm, which is part of the Staircase Join, removes nodes from a context set that would otherwise be parsed several times and yield duplicate result nodes. For example, if a context node has descendant context nodes, these can be pruned before a descendant step is traversed. But in fact pruning is unnecessary in many cases; prunable descendant nodes never occur when only child location steps are evaluated. Referring to the queries in our performance evaluation, none of the queries actually needs a Pruning of the context set. This is why we added a flag, stating if Pruning is necessary or not.

Thanks to the optimizations, the main-memory representation of an XML file occupies from 0.6 to 2.5 the size of the original document on disk (find examples in Table 4.1). The included value index just represents 12 to 18% of the main-memory data structure, and XML instances up to 13 GB have been successfully mapped into memory, still allowing efficient querying of the data.

Document	File Size	MM Size	Factor
Treebank	82 MB	182 MB	2.21
XMark	111 MB	142 MB	1.28
DBLP	283 MB	526 MB	1.86
Swissprot	1.43 GB	2.41 GB	1.69
Wikipedia	4.3 GB	5.75 GB	1.34
XMark	11 GB	10.65 GB	0.97

Table 4.1: *Main-memory consumption of BaseX*

4.1.2 Optimized On-Disk Processing

The trigger for evolving a native block storage for XPath Accelerator was twofold. First, the current reference implementation MonetDB does not scale linearly beyond the main-memory barrier. As soon as the XML data exceeds the available RAM, the performance either degrades exponentially due to extensive swapping, or the database enters an unpredictable state. Second, the latest update functionality introduced with [BMR05] essentially runs in linear time. Linear overhead for *pre* value relabeling is avoided only for page-local modifications. As soon as a whole

page must be added or removed, the page index must be updated – an operation which runs in $O(n)$ time.

TreeTank aims at efficiently querying and updating large-scale persistent XML data, e.g., as it would be required to map file system metadata to XML. We present a set of index, tuple, and block structures that allow to update XPath Accelerator encodings in $O(\log n)$ time while pushing the amount of available XML data beyond current main-memory limits. The trade-off is both the logarithmic cost to lookup a *pre* value and a potential loss of performance due to disk-based I/O.

The prototype demonstrating the feasibility of our ideas is written in Java. A rudimentary storage manager provides access to a block-oriented 64-bit storage. TreeTank currently supports an in-memory block storage for testing purposes and a random-access file-based block storage for benchmarking. To bypass the file system cache of the operating system and gain access to vast amounts of block storage, an iSCSI-based block storage is in the works. The file system cache can not exploit the tree-knowledge found in XPath Accelerator, still occupies memory and blurs potential scalability measurements because smaller XML data sets might be fully cached whereas larger XML data sets might not.

Block allocation is handled similar to XFS [WWA93]. Two B+ trees [GR93] support the dynamic allocation of single blocks or extents (multiple contiguous blocks) close to a requested address. The storage manager currently implements a simple LRU block buffer. Recent caching algorithms such as temporal-spatial caches [JDTZ05, GM05] could be plugged-in if required.

Index Structures. TreeTank employs two well-known block-based index structures to map 64-bit keys or positions to tuple addresses consisting of a 48-bit block address and a 16-bit block offset. Keys are immutable, unique, in dense ascending order, and generated by a persistent sequence as it is commonly found in database systems. Positions are volatile in the sense that they might reference different tuples over time due to updates. Note that there currently are no fulltext, path, or value index structures.

The counted B+ tree [Tat04] is a slight modification of a B+ tree. B+ trees store the key range contained in each child node. In contrast, counted B+ trees store the number of leaf values contained in the whole subtree of each child. This allows to access any element of the index by position and in logarithmic time. Updates potentially trigger expensive rebalancing operations. The Trie [dlB59] is a dense distribution of unique keys as they are frequently occurring in TreeTank. This specific key distribution allows for an index structure that does not require rebalancing [Moc07]. A set of hierarchical arrays can efficiently be queried and updated in logarithmic time because the array (i.e., block) offset of each level can be precomputed. A third index structure appearing in TreeTank, the Hash Map, is only held in main memory to speed up certain operations and can be reconstructed from a trie-based index structure at any time.

Tuple Structures. Figure 4.3 shows the core tuple and index structures of TreeTank. XPath Accelerator is persistently stored in the node list. Each XML node is stored as a node tuple (see Table 4.2) at the node-list-position equal to the *pre* value. Names and values are offloaded from the node list and separately stored as name tuples (Figure 4.2) in the name map and value tuples in the value map respectively.

The offloading of strings has four advantages. First, a very tight packaging of the frequently accessed node list results in fewer I/Os. Second, the name map can

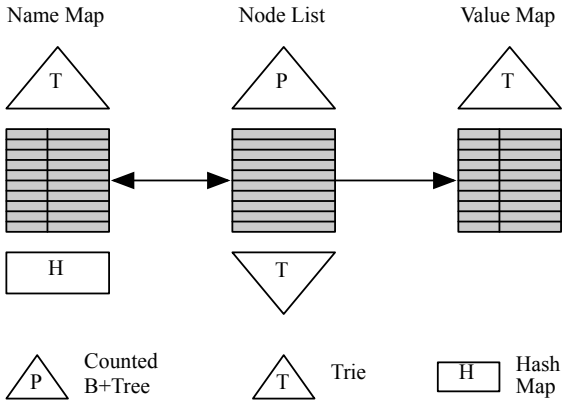


Figure 4.3: *Core tuple and index structures of* TreeTank

be kept in memory due to its small size even for very large XML data [NvdL05], leading to constant-time name-to-reference resolution. Third, filtering of node tuples according to a name can be reduced to a fast reference (integer) comparison. Fourth, the reference is usually much smaller than the string. A disadvantage is the additional cost of retrieving a value due to the additional mapping and potentially distant block address of the value tuple.

The ancestor axis is supported by an immediate reference to the parent element. The Staircase Join will therefore have to mix keyed and positional access. Since the absolute position is lost after a keyed access, the Staircase Join must always work with relative positions. A reverse access path to find the absolute position will be investigated in future work. The attribute count is stored to quickly skip attributes if they are not required for evaluation. Nevertheless, attribute nodes are kept close to the corresponding element nodes to streamline attribute-related evaluations.

Node Tuple (see Table 4.2). A node tuple can be accessed both by position and by key. Positional access is provided by a counted B+ tree. Note that a counted B+ tree does not suffer from the linear-time relabeling of *pre* values required after an update and hence offers logarithmic update behavior. Positional access is required for the Staircase Join that basically operates on *pre* values, i.e., positions. Keyed access is provided by a trie. It is required to support future index structures (such as a fulltext index) that reference specific node tuples and must not lose their context after an update.

Field	Bytes	D	E	A	T
kind	1	x	x	x	x
key	1..9	x	x	-	-
parentKey	1..9	x	x	-	-
size	8	x	x	0	0
level	1	x	x	x	x
attributeCount	1	-	x	-	-
nameReference	1..5	x	x	x	-
valueReference	8	-	-	x	x

Table 4.2: *Node tuple stored in node list. The following kinds are currently stored (denoted with 'x') for an XML node: (D)ocument, (E)lement, (A)ttribute, and (T)ext. Variable-length encodings are denoted with '..'. '0' is a constant unstored zero. '-' means not stored*

Name Tuple (see Table 4.3). A name tuple is accessed both by name and by a key stored with the node tuple. The (reverse) mapping between name and key is achieved by a hash map. This access path is required to maintain the counter (i.e., the number of occurrences of the name in the stored XML data) assigned to each name and to efficiently filter node tuples by their name. The mapping between key and name tuple is done by a trie and required whenever the name of a node tuple must be resolved.

Field	Bytes	Description
count	8	# of occurrences
name	..	UTF-8-encoded String

Table 4.3: *Name tuple stored in name map*

Value Tuple (Table 4.4). A value tuple is accessed by a key stored with the node tuple. The mapping between key and value tuple is done by a trie and required whenever the value of a node tuple must be resolved.

Field	Bytes	Description
value	..	UTF-8-encoded String

Table 4.4: *Value tuple stored in value map*

Block Structures. Figure 4.4 shows the node, name, and value block layouts. The first block shows a name or value block containing two name or value tuples. The next three blocks show an empty node block that is updated by adding two new node tuples and finally has the first node tuple removed.

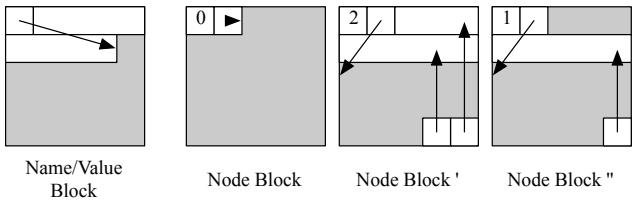


Figure 4.4: *Name/Value and Node block layout*

Node Block. Node tuples must be locatable through two different access paths. A keyed access will always yield a node block address with an immediate node block offset where the node tuple is stored. A positional access, in contrast, will return a node block address with a position relative to the block node. This position is locally resolved to an immediate node block offset with the help of an intra-node-block directory. This directory is always kept ordered by the *pre* axis and grows bottom-up according to the number of node tuples in this node block. New node tuples are appended to the last allocated node block if it still has enough contiguous space between the directory and the last inserted node tuple. If it is full, a new node block is allocated. The insertion or deletion of one node tuple affects both the counted B+ tree as well as the trie. Due to updates, a node block containing node tuples might be split or merged with another node block. This operation has an upper bound of $O(2m \log n)$, which yields $O(\log n)$; m is the number of affected node tuples. Defragmentation of node blocks from which node tuples are deleted is postponed to the next merge or split operation.

Name Block. Name tuples are written to disk like to a content-addressable storage, i.e., each name is only stored once. If a name is added, the hash map is looked up for a key associated with it. If it is found, the name tuple is located and the counter increased. The trie therefore maps the key to the name block address and an immediate name block offset where the name tuple is stored. If the name has not been stored before, the name tuple is appended to the last allocated name block if it fits. As soon as the last name block is full, a new name block is allocated. A modified name is treated like a new name. Deleting a name results in a decrease of the counter. Name tuples with zero occurrence remain accessible to support versioning. A garbage collector could free unreferenced names from time to time if required.

Value Block. Value tuples are written to disk in a log-structured fashion. If a new value tuple fits into the last allocated value block, it is appended there. If the last value block is full, a new value block is allocated. The trie maps the new key to this value block address and the immediate value block offset where the value tuple was written. The XML shredder assures that a value tuple is not bigger than a single value block by splitting larger text nodes into multiple smaller text nodes. This internal fragmentation of text nodes is not visible to the upper layers in order not to break the XPath or XQuery Data Model. A modified value is treated like a new value. Deleted values again remain accessible to support versioning.

API. TreeTank currently offers a low-level API based on a cursor pointing to a node tuple. The cursor can be moved to an absolute or relative position as well as to a given key. The cursor can retrieve all fields after locating the appropriate node tuple. Name or value strings are lazily fetched from the name and value map. New node tuples can be appended to the end of the node list.

On a higher level, a rudimentary API allows to execute a Staircase Join for a given context set on the descendant axis including simple predicate evaluation. The resulting context set consisting of ordered *pre* values can either be passed to the next Staircase Join to implement a simple XPath evaluation or materialized by fetching all fields of each context node with the cursor.

4.1.3 Evaluation Framework

We used our own Java-based benchmarking framework PERFIDIX to guarantee and facilitate a consistent evaluation of all tested systems. The framework was initially inspired by the unit testing tool JUnit [GB09]; it allows to repeatedly measure execution times and other events of interest (e.g., cache hits). The results are aggregated, and average, minimum, maximum, and confidence intervals are collected for each run of the benchmark.

Method	Unit	Sum	Min	Max	Avg	StdDev	Conf95	Runs
Query 1	ms	248	12	172	49.60	61.37	[00.00, 103.40]	5
Query 2	ms	312	49	103	62.40	20.37	[44.54, 80.26]	5
...	...	...	...	...	...	...	...	...
Query 20	ms	215	41	48	...	...	...	...
Summary	ms	17595	3228	4317	...	...	...	...

Table 4.5: *Sample output for* TreeTank *11MB XMark benchmark run by* PERFIDIX

A sample output for the TreeTank 11MB XMark benchmark is shown in Table 4.5. The whole benchmark was implemented in one Java class and each Query

as a Java method. PERFIDIX was configured to run the benchmark 5 times and only measure execution times of each method.

Our tests are based on the scalable XMark Benchmark [SWK+02] and the XML version of the DBLP database [Ley02a] (State: 2005-12-12, 283MB). We formulated six mainly content-oriented XPath queries for the DBLP data, yielding small result sets (see Table 4.6), and implemented the predefined XMark queries in our prototypes. Detailed information on the queries can be found in [SWK+02]. Single queries will be described in more detail whenever it seems helpful to understand the test results.

No	Query	Hits
1	`/dblp/article[author/text() = 'Alan M. Turing']`	5
2	`//inproceedings[author/text() = 'Jim Gray']/title`	52
3	`//article[author/text() = 'Donald D. Chamberlin'][contains(title, 'XQuery')]`	1
4	`/dblp/*[contains(title, 'XPath')]`	113
5	`/dblp/*[year/text() < 1940]/title`	55
6	`/dblp//inproceedings[contains(@key, '/edbt/')][year/text() = 2004]`	62

Table 4.6: *DBLP queries*

4.1.4 Measurement Principles

Instead of splitting up query processing into single execution steps, our test results represent the systems' overall query execution times, including the compilation of queries and the result serialization into a file. As all four systems use different strategies for parsing and evaluating queries, a splitting of the execution time would have led to inconsistent results. Table 4.7 lists the methodology of execution time measurements for each system.

System	Compile	Execute	Serialize
MonetDB	x	x	x
BaseX	hard-coded	x	x
X-Hive	x	x	x
TreeTank	hard-coded	x	x

Table 4.7: *Methodology of execution time measurements. 'x' means included in overall execution time*

The hard-coded query execution plans of BaseX and TreeTank do not allow to include the time for parsing, compiling and optimizing the queries, but we intend to extend the MonetDB engine to produce query execution plans for BaseX and TreeTank. Meanwhile, the hard-coded query execution plans are carefully implemented based on the API of each prototype to follow automated patterns and avoid "smart" optimizations that can not easily be detected and translated by a query compiler.

To get better insight into the general performance of each system, we run each query several times and evaluated the average execution time. As especially small XML documents are exposed to system- and cache-specific deviations, we used different number of runs for each XML instance. The number of executions is listed in Table 4.8.

Queries were excluded from the test when the execution time was expected to take more than 24 hours (1GB XMark Query 8, 9, 11 and 12 for X-Hive) or yielded

Document	Size	No. Runs
XMark	110 KB	100
XMark	1 MB	50
XMark	11 MB	10
XMark	111 MB	5
XMark	1 GB	5
XMark	11 GB	1
XMark	22 GB	1
DBLP	283 MB	5

Table 4.8: *Query execution times*

error messages (11GB XMark Query 11 and 12 for MonetDB). All test runs were independently repeated several times with a cold cache; this was especially important for the 11GB XMark instance which was only tested once at a time.

All tests were performed on a 64-bit system with a 2.2 GHz Opteron processor, 16GB RAM and SuSE Linux Professional 10.0 with kernel version 2.6.13-15.8-smp as operating system. Two separate disks were used in our setup (each formatted with ext2). The first contained the system data (OS and the test candidates), the second the XMark and DBLP input data and the internal representations of the shredded documents for each system. The query results were written to the second disc. We used MonetDB 4.10.2 and X-Hive 7.2.2 for testing.

We compare the in-memory BaseX with MonetDB to analyse the impact of the optimizations applied with BaseX. Then we compare the disk-based TreeTank with X-Hive to confront natively persistent XPath Accelerator to Persistent DOM. Finally, we compare TreeTank with MonetDB to verify our assumptions about the difference of an in-memory and a disk-based system. The comparison of TreeTank and BaseX is not explicitly mentioned but can be deduced from the presented figures and analysis.

4.1.5 Benchmark Results

Figure 4.5 gives an overview on the scalability of each system and the average execution times of all XMark queries and XML instances. First observations can be derived here: MonetDB and BaseX can both parse XML instances up to 11GB whereas TreeTank and X-Hive could easily read and parse the 22GB instance. The execution times for the 11GB document increase for TreeTank and MonetDB as some queries generate a heavy memory load and fragmentation. The most obvious deviations in query execution time can be noted for the Queries 8 to 12, which all contain nested loops; details on the figures are following.

An aggregation of the 20 XMark queries is shown in Figure 4.6, summarizing the logarithmic values of all XMark query averages. All systems seem to generally scale linearly on the tested queries. MonetDB consumes an average time of 38ms to answer a query, mainly because input queries are first compiled into an internal representation and then transformed into the MonetDB-specific MIL language. Though, the elaborated compilation process pays off for complex queries and larger document sizes.

BaseX and MonetDB – XMark. Obviously, BaseX yields best results for Query 1, in which an exact attribute match is requested. The general value index guarantees query times less than 10ms, even for the 11GB document. MonetDB is

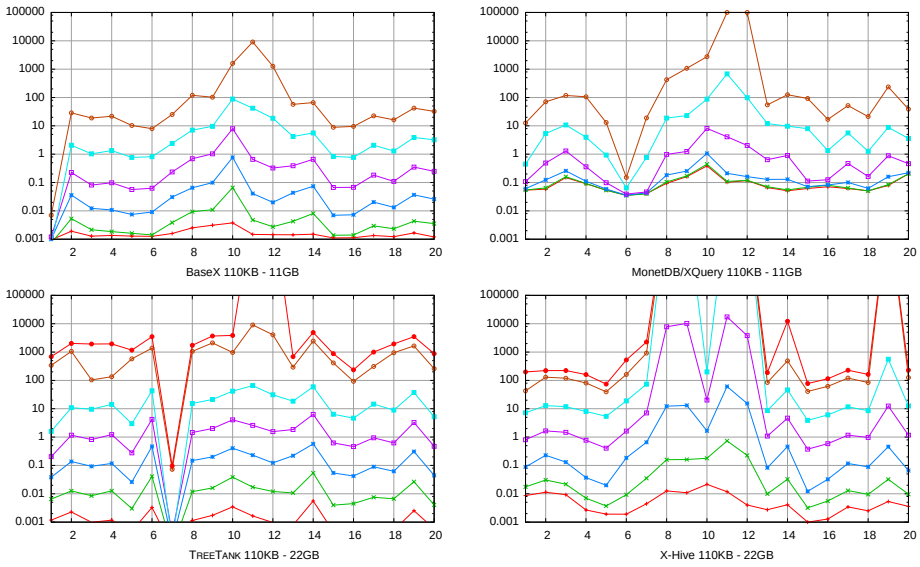


Figure 4.5: *Scalability of the tested systems (x-axis: XMark query number, y-axis: time in sec)*

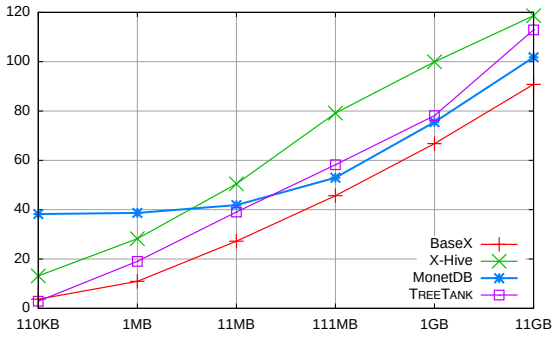


Figure 4.6: *Logarithmic Aggregation of all XMark queries (y-Axis: Sum of i=1..20 of* $\log avg_i$ *in ms)*

especially good at answering Query 6, requesting the number of descendant steps of a specific tag name. MonetDB seems to allow a simple counter lookup, thus avoiding a full step traversal.

The top of Figure 4.7 relates the query times for MonetDB and BaseX and the XMark query instances 111MB, 1GB, and 11GB. For the 111MB instance, BaseX shows slightly better execution times than MonetDB on average which is partially due to the fast query compilation and serialization. The most distinct deviation can be observed for Query 3 in which the evaluation time of position predicates is focused. Results for Query 8 and 9 are similar for both systems, although the underlying algorithms are completely different. While MonetDB uses loop-lifting to dissolve nested loops [BGea06], BaseX applies the integrated value index, shrinking the predicate join to linear complexity. The respective results for the 1GB and 11GB instance underline the efficiency of both strategies.

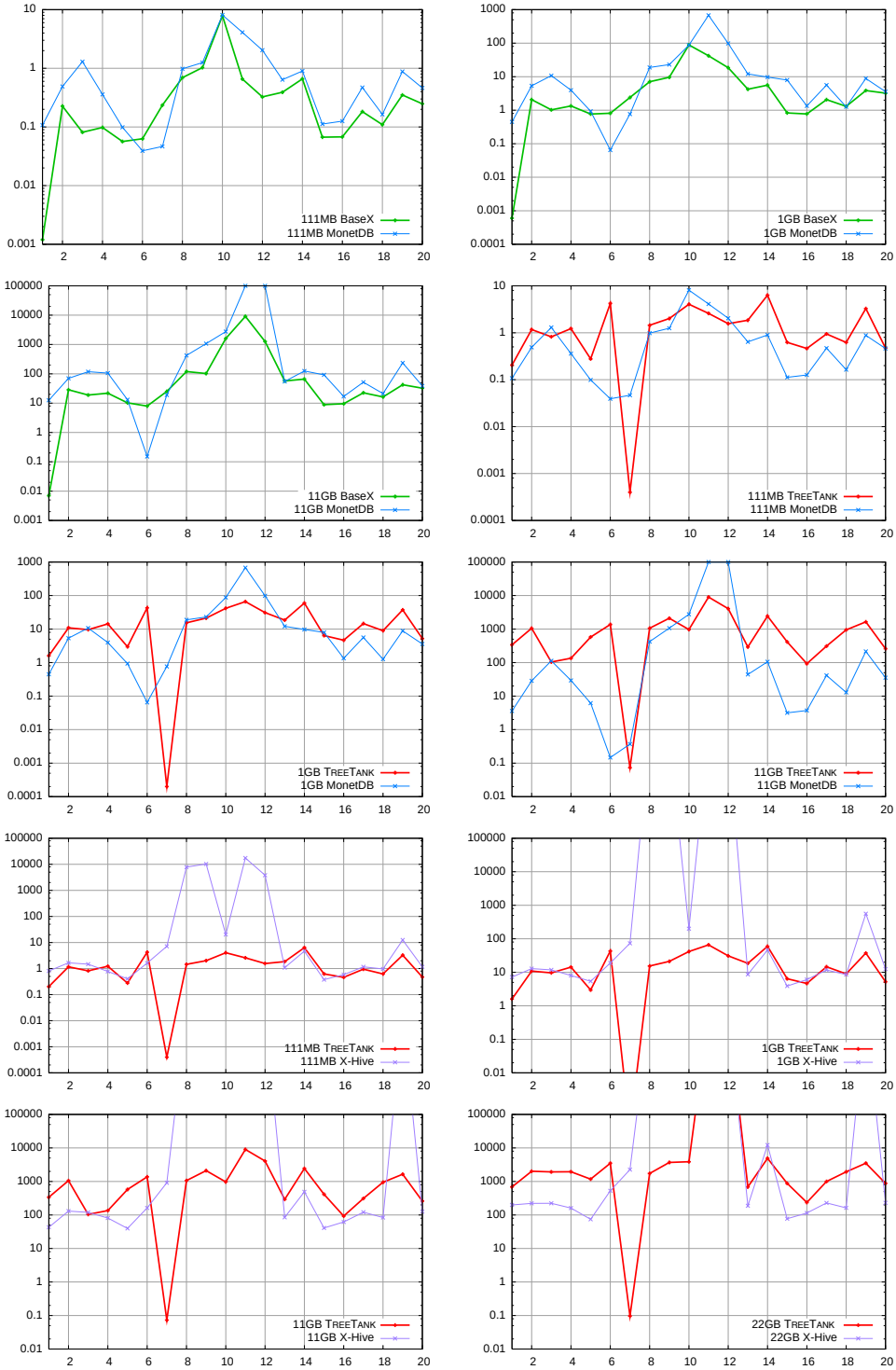


Figure 4.7: *Comparison of the candidates (x-axis: XMark Query number, y-axis: time in sec)*

The Queries 13 to 20 yield quite comparable results for MonetDB and BaseX. This can be explained by the somewhat similar architecture of both systems. The most expensive Queries are 11 and 12, which both contain a nested loop and an arithmetic predicate, comparing double values. To avoid repeated string-to-double conversions, we chose to extract all double values before the predicate is actually processed. This approach might still be rethought and generalized if a complete XQuery parser is to be implemented.

TreeTank and X-Hive – XMark. Both TreeTank and X-Hive store the XML data on disk and consume only a limited amount of memory for the buffer. They differ however in the handling of query results. TreeTank materializes all intermediate query results in main memory and only writes them to disk after successfully executing the query. This results in the extensive alluded memory consumption and fragmentation for XML instances bigger than 1GB. X-Hive immediately writes the results to disk.

We expected the systems to show query execution times in the same order of magnitude as both are essentially disk bound. The effects of block allocation, buffer management, and the maturity of the implementation (optimizations) as well as the different materialization strategy were assumed to have a minor impact. TreeTank was expected to have a small constant advantage because the queries are not compiled and a disadvantage because it currently is fully synchronous, i.e., disk and CPU intensive phases do not overlap and hence waste resources. Both systems are equal with respect to value or fulltext indexes: while TreeTank does currently not support them, X-Hive was configured not to create them.

In Figure 4.7 (111MB and 1GB TreeTank and X-Hive), our expectations are only partially confirmed. With Query 7, TreeTank can take advantage of the name occurrence counter (comparable to a reduced variant of the MonetDB path summary) and quickly calculates the count of a specific element name. Note that this would take longer if the count would be applied on a subtree, starting at a higher level. Queries 8 to 12 also clearly deviate. Here, the execution time of the query is no longer simply disk-bound but dependent on the decision of the query compiler and execution plan. TreeTank uses a hash-based join by default whereas X-Hive probably executes a cross-join or another unapt join variant. Note that TreeTank does not estimate the sizes of the sets to join but just picks the one first mentioned in the query.

The memory allocation and fragmentation resulting from the full in-memory materialization of intermediate results with TreeTank has an unexpected though major impact on the overall performance which is also aggravated by the garbage collection of Java. The linear scalability beyond 1GB is therefore slightly impaired. An immediate lesson learned is to switch to the same serialization strategy as it is employed with X-Hive.

TreeTank and MonetDB – XMark. We consider MonetDB to be the reference XMark benchmark in-memory implementation. We expected TreeTank to perform an order of magnitude slower because it is disk-bound. The results confirm the expectations and only step out of line for the Queries 7 (see discussion of TreeTank and X-Hive) and 10 to 12. In Figure 4.7, the plots for 111MB, 1GB, and 11GB for TreeTank and MonetDB consistently show a surprising result for the queries 10 to 12 where MonetDB performs worse than TreeTank. The same argument, i.e., the query execution plan, applies as with X-Hive. The main-memory consumption and fragmentation of MonetDB to materialize intermediate results is a supporting argument.

BaseX – DBLP. The query performance for BaseX was measured twice, with the value index included and excluded. The results for BaseX and the Query 4 and 5 are similar for all tests as the contains() function demands a full text browsing for all context nodes. The index version of BaseX wins by orders of magnitude for the remaining queries as the specified predicate text contents and attribute values can directly be accessed by the index. The creation of large intermediate result sets can thus be avoided, and path traversal is reduced to a relatively small set of context nodes.

TreeTank – DBLP. Figure 4.8 summarizes the average execution times for the queries on the DBLP document. The lack of a value and fulltext index forces TreeTank to scan large parts of its value map to find the requested XML node. This holds for all six queries and results in near-constant query execution time for all DBLP Queries.

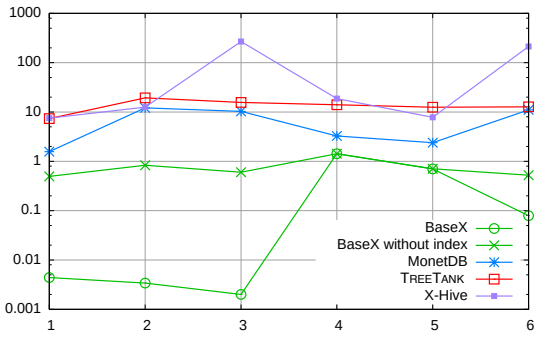


Figure 4.8: *DBLP Execution Times (x-axis: DBLP Query number, y-axis: time in sec)*

4.1.6 Conclusions and Outlook

In this section we presented two improvements backed with their corresponding prototypes to tackle identified shortcomings with XPath Accelerator. The performance and scalability of these two prototypes were measured and compared to state-of-the-art implementations with a new benchmarking framework.

The current prototype of BaseX outperforms MonetDB. The tighter in-memory packaging of data as well as the value index structure stand the test. Though, the presented evaluation times are still subject to change as soon as a more elaborated query compilation is performed. TreeTank can efficiently evaluate XML instances beyond the main-memory barrier. The cost of making XPath Accelerator persistent is paid-off for larger XML data sets. TreeTank introduces logarithmic overhead to locate a node tuple by position or by key. This overhead is negligible compared to the overhead resulting from disk I/O and largely compensated by caches. We regard our paradigm shift away from constant time array lookups, as found in in-memory XPath Accelerator implementations, a good trade-off with disk-based implementations because it allows superior update behavior and scalability.

The Persistent DOM concept as implemented with X-Hive shows a scalability and query performance comparable to XPath Accelerator. This is a surprising result since former versions of X-Hive did not even scale beyond the 1GB XMark document [BGea06].

BaseX still lacks features such as the support of several documents, namespaces or the storage of some specific XML data, including comments or processing instructions. Node information of this kind will lead to an extension of the existing data structure. Moreover no schema information is evaluated yet, and some effort has still to be invested in implementing or integrating a complete XPath and XQuery implementation.

However, the code framework was carefully designed to meet the requirements of the XPath and XQuery specifications. A major focus will next be set on further indexing issues, supporting range, partial and approximate searches. Besides, some effort will be put on fulltext-search capabilities, expanding the prototype to support flexible fulltext operations as proposed by the W3C [AYBB+06].

4.2 Node-Level Granularity

The implementation of TreeTank serves as a proof-of-concept of the specification provided in Section 3.1. An early implementation as described in Subsection 4.1.2 was used for performance analysis. As soon as the analysis proved that we are in line with our claims, we evolved the implementation described in this section to better support research on interfaces (see Chapter 5), applications (see Chapter 6 and Chapter 7) as well as the aspect of distribution. Of course, this latest implementation keeps all runtime characteristics of the early implementation and consists of three well-separated layers:

Node Layer Each node contains the information about its content and position in the overall structure. The *Parent/First Child/Left Sibling/Right Sibling* tree encoding was chosen to assure this.

Page Layer The (transient) position of each node in the tree is decoupled from its (stable) position in the storage organized as pages.

Transaction Layer This layer is directly accessible to the applications and handles a single write transaction (for node modifications) as well as multiple parallel read transactions (on any version).

4.2.1 Node Layer

Figure 4.9 lists all implemented node types. 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 `Node` in Figure 4.9, is valid for one node in all versions. It satisfies the following three requirements:

1. The unique identifier acts as a direct reference to 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 is immutable and not reassigned to new nodes even after deletion.

Besides the unique identifier, all nodes hold a reference to their immediate parents. Additional structural nodes such as `ElementNode` and `TextNode` hold a reference to their immediate left and right siblings as well as their first child. All of these fields are denoted as structural attributes and colored cyan in Figure 4.9.

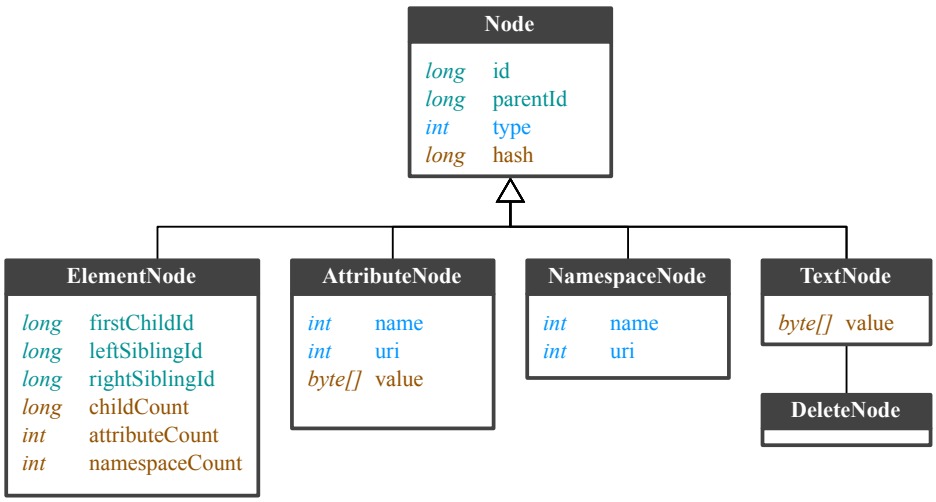


Figure 4.9: *Node types implemented by* TreeTank

Regarding the content of a node, each node holds a specific type key equivalent to the XSD type. 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. In addition, name and URI of `NamespaceNode`, `AttributeNode` and `ElementNode` are pointers to this central string store as well. The pale blue color of the attributes in Figure 4.9 denote these referential attributes.

Content which cannot be referenced, is stored directly. One example is the hash value. The tree structure generates the hash with the help of the recursive relationship between nodes. Therefore, each hash value of a concrete node guards the integrity of its corresponding subtree. This allows to track changes in our structure without expensive diff-computations on subtrees. Other attributes representing direct content are the count of associated nodes such as children, attributes and namespaces in `ElementNode` as well as the concrete text denoted as value in `TextNode` and `AttributeNode`. These fields are called content attributes and colored brown in Figure 4.9.

As already mentioned, the tree encoding consists of exactly four pointers to the unique identifiers of the immediate parent, first child, left sibling, and right sibling. Figure 4.10 shows such an encoded XMark [SWK+02] instance: Since `ElementNode` and `TextNode` are structural, they embody the structure of the tree in their immediate neighborhood, e.g., each *category* node holds a reference to its parent *categories*. Additionally, the first child *name* is referenced as well as, if they exist, the related left and right sibling nodes *category*. Attributes such as *id* are not treated as structural nodes due to their special association to `ElementNode`. The tree encoding uses specialized pointers for those associated nodes. The nodes in Figure 4.10 map, according to their colors, to the different node types explained before.

When it comes to modifications, note the insertion of the new child node *namerica* as a child of the existing *regions* node in Figure 4.10. This node should be inserted

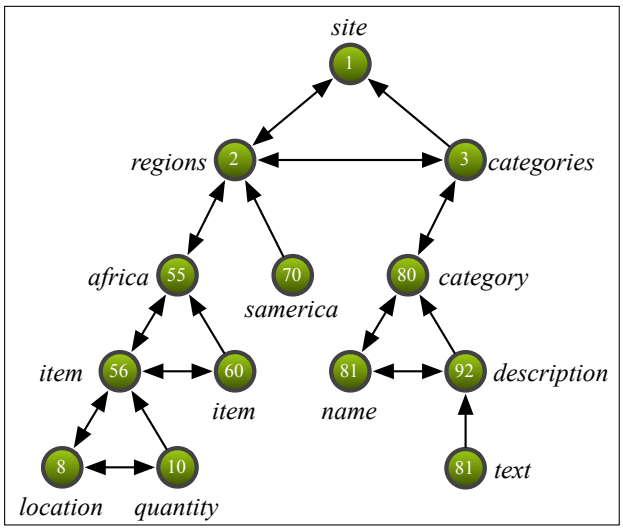


Figure 4.10: *Example Encoding of XMark-Instance in* TreeTank

as a left sibling node of *samerica*. This results in the following adaptions:

1. The child counter of the parent is increased.
2. The hash value of the ancestors are adapted.
3. The right sibling pointer of the new *namerica* node is redirected to the existing *samerica* node.
4. The left sibling pointer of the new *namerica* node is redirected to the existing *africa* node.
5. The right sibling pointer of the existing *africa* node is redirected to the new *namerica* node.
6. The left sibling pointer of the existing *samerica* node is redirected to the new *namerica* node.

As such, the adaptions on the encoding only take place on the immediately related nodes. Even if we insert big subtrees below 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:

1. The child counter of the parent is decreased.
2. The hash value of the ancestors are adapted.
3. The first child pointer of the parent is redirected to the existing *category* node which acted as a right sibling of the node to be removed.
4. 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 the insertion and removal operations only have a minimal impact on the overall structure. Due to the locality of our approach, no global adaption takes place. Furthermore the locality of our approach offers us great possibilities to provide iterators based on the pointers and therefore 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., `CommentNode` 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 directly on the nodes. Our node layer therefore not only provides fine granular versioning but also extensibility and flexibility for future work.

4.2.2 Page Layer

The second layer is the page layer. Since the node layer itself has no functionality to version any data, the page layer takes care about the handling of different versions. However, the versioning functionality is not easily adaptable to common sequential paging architectures where ensuring constant access and minimal rearranging of data with respect to consecutive changes is mandatory. Therefore, we introduce a more complex structure which offers the same access to all data with minimized redundancy and with a copy-on-write architecture. This results in a tree structure of pages motivated by the page layout of ZFS [BAH+03]. Figure 4.11 illustrates the chosen architecture. The pages have different attributes and intentions.

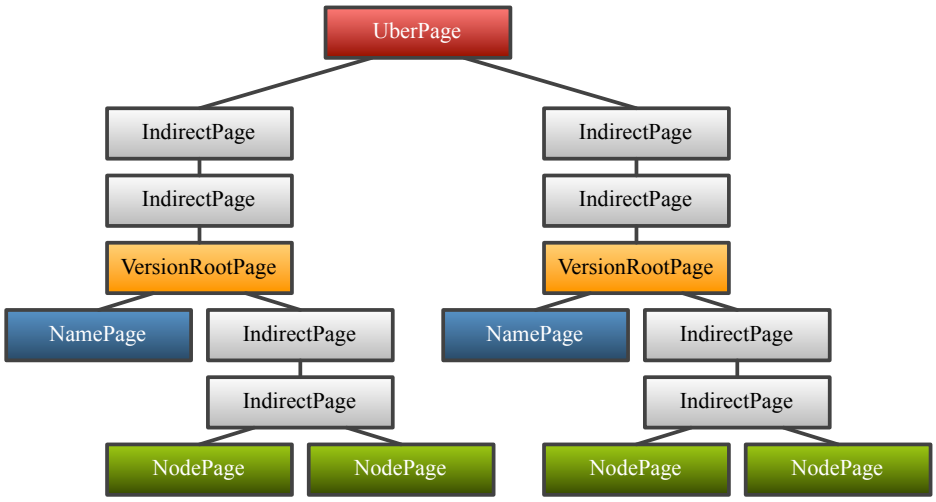


Figure 4.11: *Page Layer architecture. Each box denotes a separate page*

UberPage The `UberPage` is the main entry point for each store and retrieval process. Therefore, it is the only page which stays synchronized at all times for ensuring the integrity for all versions. It holds the global version counter as well as direct references to the `IndirectPage` and hence indirect references to the `VersionRootPage` and the `NodePage`. Note that the `UberPage` is comparable to the *UberBlock* in ZFS [BM04].

IndirectPage The `IndirectPage` acts as a container for pointers to other pages.

The `IndirectPage` links the `UberPage` with the `VersionRootPage` as well as a `VersionRootPage` and its specific `NodePage`. Both times, the `IndirectPage` are organized in a static tree of height 5. Based on a pointer set holding 128 pointers in each `IndirectPage`, we can handle 128^5 versions and nodes at the moment.

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

NodePage The `NodePage` finally contains all nodes. Therefore, `NodePage` are always leafs in the page layer. Starting from a `VersionRootPage`, the unique identifier of a node leads us to the related `NodePage`. `NodePage` follow the copy-on-write principle. Hence they do not offer possibilities to rearrange nodes within the page.

NamePage The `NamePage` is the central storage of common tag names, URIs and types. Regarding its architecture, it acts as a flexibly sized hash map. Since tag names mainly consist out of a fixed set related to common XML instances, we can decrease the overhead of storing atomic strings. Instead, we reference the strings in the `NamePage` related to the corresponding version and point to them through unique keys in the related nodes.

Note that the Page Layer can be adapted to support full, incremental, differential, or even SlidingSnapshot versioning algorithms. A `NodePage` in the first version stores all nodes inserted in the first version. Afterwards, only the modified nodes are stored in the `NodePage` associated to the first `VersionRootPage`. Regarding incremental versioning, all changes are stored in the `NodePage` associated to the last `VersionRootPage`. This results in a suitable set of `VersionRootPage` related to each transaction.

Every transaction must hold the suitable set of `VersionRootPage` to reconstruct the structure. This set depends on the current versioning algorithm. For incremental versioning, the set of `VersionRootPage` 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 `VersionRootPage`, each node is referenced in the same manner over the layer of `IndirectPage`. This results in the same performance regarding all versions and all nodes. `DeleteNode` are inserted during removal operations to ensure the correct reconstruction of the tree distributed over multiple `VersionRootPage` and their subtrees.

The independent node dereferencing regardless of its versioning and storage approach is founded on the unique identifier of the node. This unique identifier not only acts 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 system, this referencing architecture fits perfectly to our copy-on-write approach. If one `NodePage` reaches its maximum number of nodes, upcoming nodes are stored to the next `NodePage`.

This results in different distributions of nodes across a `NodePage` based on their order of insertion. Figure 4.12 shows the same tree stored on different set of `NodePage` since the structure is inserted into our system in different orders. However, based on

the unique identifier of each node, each `NodePage` can be dereferenced in the same manner and in the same time. Since forecasting all modifications between versions 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 copy-on-write approach, we are further avoiding common problems such as fragmentation or page reorganization. On the other hand, we introduce a constant, or generally speaking, logarithmic node access overhead. But we strongly believe this is worth it, especially when it comes to flash storage.

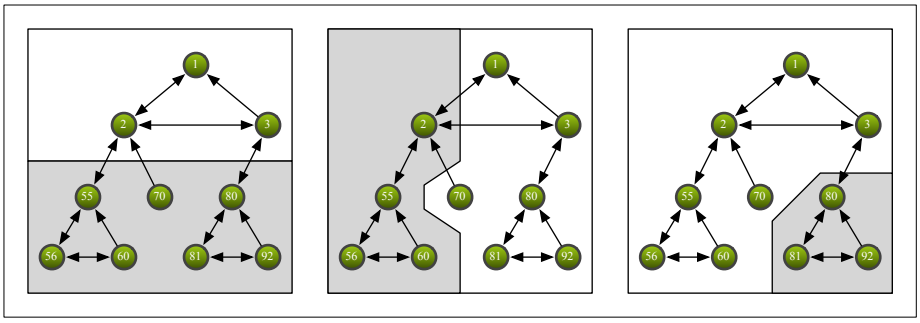


Figure 4.12: *Three different examples of how to partition the same set of nodes into two different* `NodePage` *instances*

4.2.3 Transaction Layer

The third layer is the Transaction Layer. This layer acts as the bridge between the Node Layer and Page Layer, because 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 a single one at a time while letting read access work in parallel. This results in a distinction between `ReadTransaction` and `WriteTransaction`.

`ReadTransaction` is the common way to work with TreeTank when it comes to query and retrieval operations. Each `ReadTransaction` is bound to a version or time stamp to get 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 currently bound to, e.g., the move to a left sibling is not possible if the cursor points to an `AttributeNode`. The `ReadTransaction` is not only independent regarding its version but also regarding threads. Each access to the data is thread-safe which results in a highly scalable system.

In contrast to the `ReadTransaction`, the `WriteTransaction` encapsulates modifications. These modifications cover both content-related and structural updates. Each `WriteTransaction` has the same operations common for `ReadTransaction`: They navigate the tree based on normal cursor move operations. In addition, they offer methods to insert new elements. Notably, the insertion of structural elements either takes place as insertion as a first child or insertion as a right sibling. Node insertions result in:

- the creation of a new node. This new node gets its unique identifier during creation time and is appended at the end of a NodePage with free space.
- the adaption of the neighborhood. The related nodes are not recreated. Instead, only the pointers are adapted. Nevertheless, a fresh copy of the nodes is stored within a new `NodePage` related to the current `VersionRootPage`.
- the temporal persistent storage of the touched `NodePage`. These pages are cached in a persistent transaction log. This transaction log is flushed to a new version 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 version visible to other transactions.

Figure 4.13 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 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 ID 76 and a new subtree is inserted as the right sibling. Again, the dotted lines show the pointers to be adapted.

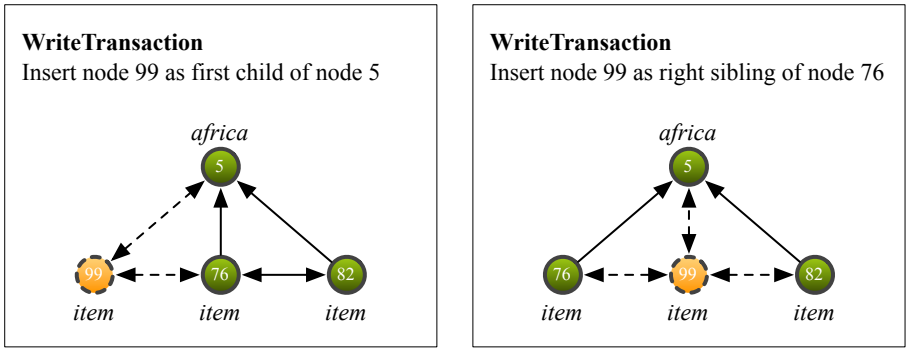


Figure 4.13: *Insertion of nodes as first child and as right sibling*

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 agnostic of 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.

4.2.4 Layer Interaction

Together, these three layers implement the main functionality of our system. An example of the interaction of all layers is shown in Figure 4.14.

Already existing data in version 1 is retrieved with the help of a `ReadTransaction`. The red area denotes this transaction which is bound to the pages under the `VersionRootPage` with version 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`

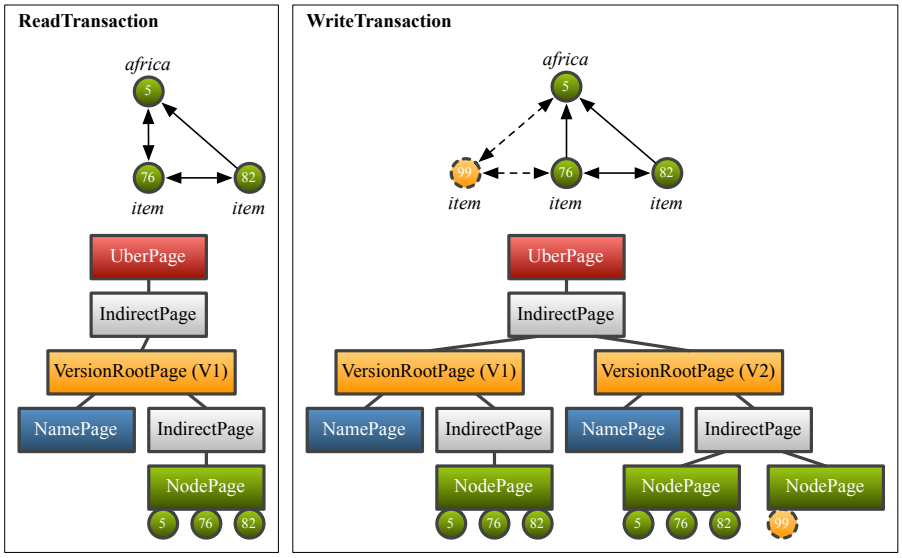


Figure 4.14: *Interaction of the three layers during an insertion operation*

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 `NodePage`, under a new `VersionRootPage` which represents the new version 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, we included caching in the `ReadTransaction` which results in the temporal storage of the actual page regarding the requesting transaction. Due to the binding of transactions to `VersionRootPage`, 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., `CommentNode` with only minimal effort. Such an extension only results 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 the node level we have the ability to make constant validations 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 and respect 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 `WriteTransaction` 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 `NodePage`.

4.2.5 Scalability Verification

To verify the scalability of our second-generation implementation of TreeTank, we insert XMark [SWK+02] instances of two different sizes multiple times into our system. Each insertion results in a version. To keep the number of nodes constant per version, we remove the data from the old version as part of every new insertion operation.

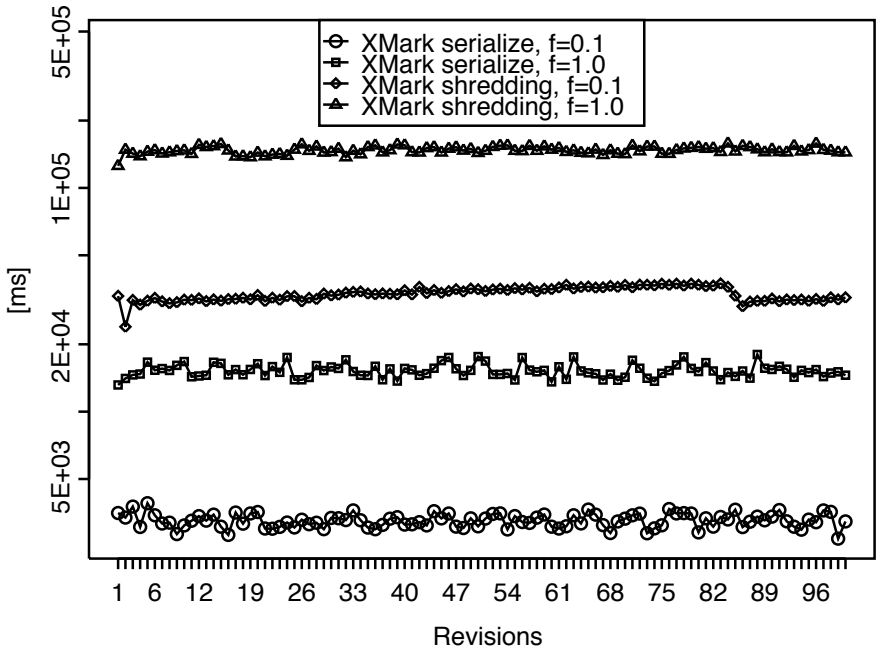


Figure 4.15: *XMark shredding and serialization*

Figure 4.15 shows the result. The shredding represents the insertion process based on a `WriteTransaction` while the serialization represents the retrieval process based on a `ReadTransaction`. We see that the implementation still scales with the size of the data as well as with the insertion and retrieval time. The insertion operation always takes 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 logarithmic modification overhead becomes clearly visible within our random insert verification. Nodes of type `ElementNode` 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 versions are created.

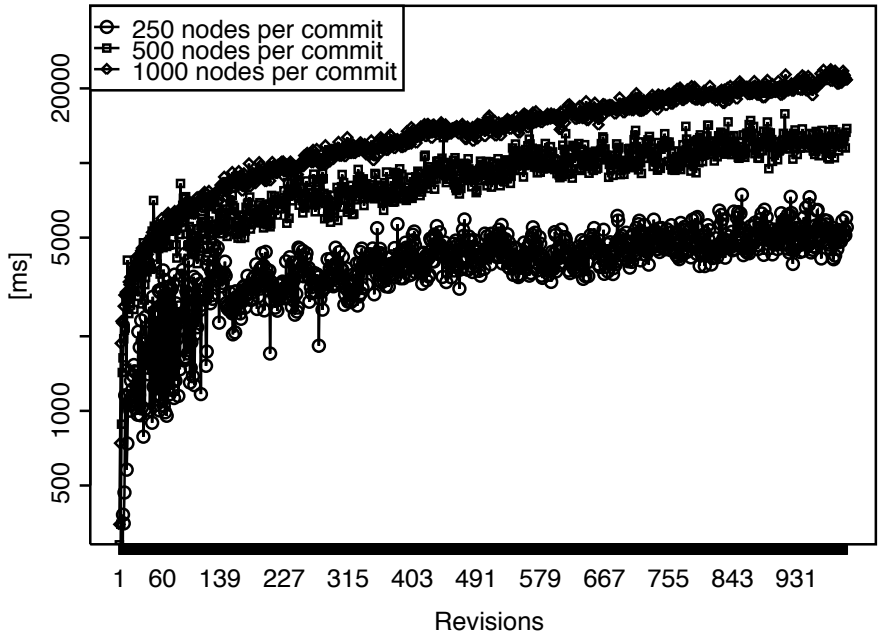


Figure 4.16: *Random insert times*

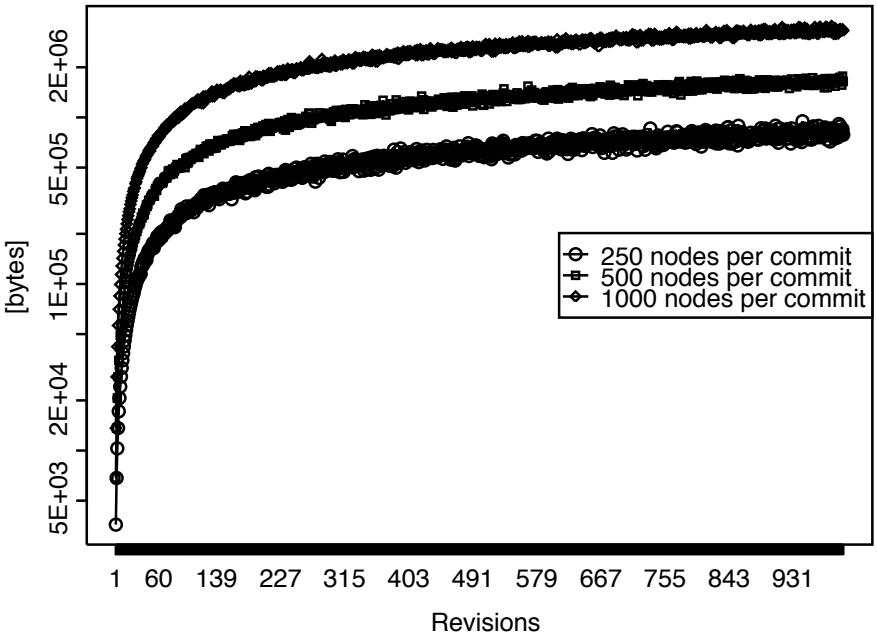


Figure 4.17: *Random insert space*

Figure 4.16 and Figure 4.17 show the insertion time and space requirements. Since the insertion takes place on a constantly increasing structure, the dereferencing of the sibling and parent nodes for adaption and the storage of the new and modified pages needs logarithmic effort. 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.

The copy-on-write approach results in a logarithmic adaption of the data regarding each incremental version. This is based on our Page Layer, which stores all pages with nodes where the pointers have to be adapted as well as the corresponding `IndirectPage` of a new and modified `NodePage`. Since this random insert operation results in huge deltas between two versions, logarithmic scaling of our storage satisfies our aim of a versioned storage.

Our second-generation implementation of TreeTank offers the combination of XML and versioning on the node level. The key aspect of the proposed architecture are the different layers in our architecture which offer flexible adaptions to different workloads due to their independence from each other. Based on our encoding and the copy-on-write paging system, we are able to version any XML instance with respect to scalability and performance.

4.3 Tools

The value of good tools can not be overemphasized. They support the researcher during implementation of his ideas, be it as a production-ready code or just a quick mockup or proof-of-concept. Essentially, they save time and provide quick insights into new ideas. In the course of our work on Java-based native XML storages, we evolved three tools, two of them readily available for anyone as open source:

jSCSI described in Subsection 4.3.1: A Java iSCSI implementation which provides scalable block-level access to Java-based native XML storages. jSCSI is released as an open source project under the BSD 3-Clause License [Ope99] and is available from `http://jscsi.org` [KG06a].

PERFIDIX described in Subsection 4.3.2: A Java benchmarking tools which provides a convenient yet scientific-ready way to analyze the performance of Java-based native XML storages. PERFIDIX is released as an open source project under the BSD 3-Clause License [Ope99] and is available from `http://perfidix.org` [KG06b].

VISIDEFIX described in Subsection 4.3.3: A block access monitoring tool which provides realtime monitoring of Java-based native XML storages.

4.3.1 jSCSI

Introduction

Accessing persistent storage from Java usually means talking to the file system through the frugal Java file system abstraction, i.e., the File class. Currently, there is no platform-independent way to directly talk to a single local or remote storage device, yet alone a device pool. jSCSI aims to fill this gap by implementing the

iSCSI protocol right in Java. We believe that, firstly, Java is mature enough to cleanly implement well-performing low-level storage protocols and, secondly, that it would be very convenient to plug a terabyte-sized iSCSI RAID into the local network and immediately connect to it from any JVM. Eventually, we need such a low-level device access for our TreeTank and SlidingSnapshot implementations.

The iSCSI protocol defines how a client (iSCSI initiator) accesses a block device on a server (iSCSI target) over a TCP/IP network. It is inspired by the existing SCSI protocol used to access local hard drives or other devices in a block-oriented fashion. Being standardized in April 2004 with RFC 3720 [SMS+04], it was quickly adopted, not least because it is believed to offer a better price-performance ratio and fewer infrastructure changes than competing solutions such as fibre channel [Ada03]. Furthermore, recent research indicates that user-level iSCSI initiators can improve performance considerably [SH05]. The main reason, as argued by the authors, is due to the reduced copy-overhead induced by the user-space to kernel barrier.

jSCSI includes a Java iSCSI initiator implemented by [Wil07], a Java device activity monitoring tool and a preliminary iSCSI backend for the widely used full-text search engine Lucene [Apa97]. Future jSCSI releases shall come with an adaptive storage pool inspired by Sun Microsystem's ZFS [BM04] as well as a more elaborate jSCSI initiator and jSCSI backend for Lucene. Releasing jSCSI under the BSD 3-Clause License [Ope99] to the open source community will allow a bigger audience to work with devices out of Java as they would work with files [KG06a].

Implementation

The first release of jSCSI provides a simple interface for a device, i.e., `Device`, as listed in Figure 4.18. A `Device` implementation must comply with the following semantics: Multiple threads can concurrently call the `read(...)`, `write(...)`, and `getX()` methods. Each method call of one thread is executed synchronously. Operation queueing and reordering is the task of the device implementation whereas caching is the responsibility of the upper layers.

```
public interface Device {
  public void open();
  public String getName();
  public int getBlockSize();
  public long getBlockCount();
  public void read(final long address, final byte[] buffer);
  public void write(final long address, final byte[] buffer);
  public void close();
}
```

Figure 4.18: *Java `Device` Interface*

The jSCSI 1.0 initiator implements the `Device` interface and binds each device to one iSCSI target. The jSCSI 1.0 storage pool is a small extension to map a device to a striped or mirrored RAID currently consisting of two or four devices for improved performance or reliability. The initiator can be configured to establish multiple sessions to various targets. Each session uses exactly one TCP connection and operates synchronously. The initiator supports the login operational negotiation as well as the full feature phase to configure the behavior of the session and to transmit

data. A set of parsers and serializers together with a state machine per session and connection assure the proper execution of the iSCSI protocol.

Besides the basic functionality for accessing an iSCSI device from Java, JSCSI 1.0 also comes with a device monitoring tool VISIDEFIX (described in Subsection 4.3.3) that allows to remotely visualize the activity of multiple JSCSI initiators. The JSCSI 1.0 initiator therefore sends a dump of the `Device` interface method calls (excluding the buffer contents) to a monitoring instance over a TCP connection. The monitoring instance running in Eclipse [Ecl01] then interactively displays the read and write touches on the device in an Eclipse view.

Finally, we implemented a backend for Lucene, i.e., a `DeviceDirectory`. This allows to store a Lucene full-text index directly on a raw iSCSI target. The Lucene example demonstrates how JSCSI can be used, what performance it achieves, and how complex the setup and maintenance is compared to a filesystem backend. The current `DeviceDirectory` works according to the log-structured, i.e., copy-on-write principle only appending new blocks at the end.

Preliminary benchmarking output measured with PERFIDIX (described in Subsection 4.3.2) for the device and filesystem backend of Lucene as well as a few read and write operations on a single or multiple targets with both the Java and Open-iSCSI initiator [Ope05] are listed in Table 4.9. The numbers give a first impression of the performance available with an average computer and networking equipment. Final results with a detailed analysis of all use cases are left to future work. Note that JSCSI uses a simple LRU cache but no prefetching and that cache hits are only available for searching a Lucene index [Luc07].

Title	Min	Max	Avg	Stddev	Conf95
jSCSI Read 40kB	2	14	3.34	1.32	[3.08, 3.60]
Open-iSCSI Read 40kB	4	15	5.29	1.32	[5.03, 5.55]
jSCSI Read 400kB	22	24	22.48	0.52	[22.38, 22.58]
Open-iSCSI Read 400kB	5	16	6.79	3.19	[6.17, 7.41]
jSCSI Write 8MB 1 Disk	3889	6987	4347.55	398.42	[4322.86, 4372.24]
jSCSI Write 8MB RAID 0	2346	4947	2803.46	330.12	[2783.00, 2823.92]
jSCSI Write 8MB RAID 1	4663	9447	5313.72	487.65	[5283.50, 5343.95]
Lucene Build File Index	523	1270	676.40	153.80	[609.00, 743.80]
Lucene Build jSCSI Index	6363	7889	6927.95	468.25	[6722.73, 7133.17]
Lucene Search File Index	2	68	7.05	14.09	[0.87, 13.23]
Lucene Search jSCSI Index	5	10	9.45	1.36	[8.85, 10.05]

Table 4.9: *Benchmark results including minimum, maximum, average, standard deviation, and the 95% confidence intervals. The unit is in ms and the benchmark was executed 1000 times*

Conclusions and Outlook

Several improvements are planned for JSCSI 2.0. The Device interface currently comes with a contract for synchronous interaction on a per-thread basis. Asynchronous, i.e., non-blocking I/O semantics will allow applications to scale better due to a more efficient use of available CPU resources. There will also be an abstract `Device` implementation coming along with queuing and prefetching support and other common features for all available devices.

The JSCSI 2.0 initiator will support multiple pending operations per connection. This works according to the pipelining principle that allows to substantially in-

crease the throughput. Another upcoming feature is the security negotiation phase and user authentication. Timers and keep-alive pings will assure that the connection is not torn down unexpectedly. Multiple TCP connections per session will be implemented when the common iSCSI targets (such as the Enterprise iSCSI Target [iSC04]) will start supporting this feature. Note that multiple connections can improve the resilience of a session as well as the throughput because of the multipathing effect [DT05]. Finally, we continue working on optimization toward smaller memory footprint as well as reducing garbage collection overhead and CPU consumption.

The JSCSI 2.0 storage pool is planned to be extended to a full-fledged storage pool similar to ZFS's storage pool. The main reason for duplicating the pool functionality in Java is the ease of rapid prototyping coming along with Java, especially compared to in-kernel development. One of the next features to develop is the ability to balance writes across multiple devices according to the device space usage, activity, and latency statistics.

The use of Java for an iSCSI implementation disclosed one minor drawback as it does not support unsigned primitive types which are extensively used with the iSCSI protocol data units. Wrapping these values into Java signed ones increases the code clutter and slightly reduces the performance due to additional tweaks of the sign bit.

Given that the JSCSI initiator currently only allows to have one pending synchronous operation and only supports a simple LRU cache without prefetching functionality, we are confident that pipelining, asynchronous operation, and prefetching will significantly boost performance in upcoming releases. All technologies in use today for improving storage device performance can be applied to Java in a portable way, opening new opportunities in protocol research and education. As such, JSCSI should provide a powerful means to quickly implement new ideas instead of bringing them into the kernel of an operating system in a tedious and error-prone development effort.

To the best of our knowledge, JSCSI is the first Java iSCSI initiator available. For the first time, the developer can access a storage device right from Java in a portable, efficient, and easy-to-use way. The rapid prototyping approach available with Java not only allowed us to implement a first version within a few man-months time but also gives us the chance to experiment with design alternatives and quickly add new functionality. This turned out to be extremely valuable, especially for research and educational purposes. Preliminary benchmark results of JSCSI 2.0 show that multithreading enables a Java-based iSCSI implementation to outperform or work on par with a native C-based iSCSI implementation [GBW09].

4.3.2 PERFIDIX

Introduction

In the course of a research project related to prototyping Java-based native XML databases, our team was repeatedly faced with the question which algorithm or data structure performed better for a given workload. Complexity analysis and big-O notations do provide theoretical boundaries and general trends. However, implementations often differ a lot in practice due to implementation details, optimizations, and CPU or RAM availability.

After a tedious array of rudimentary benchmarking efforts ranging from simple hand-coded time measurements to expensive professional profiler sessions, we soon decided to design and implement our own tool which would allow for convenient and consistent benchmarking of arbitrary Java code. Convenience should be guaranteed by a seamless integration with Java language features, existing open source development environments, and tools such as Java annotations, Eclipse [Ecl01], and Ant [Apa99]. Consistency should be assured by providing persistent and sound statistical evaluations in several output formats. Releasing PERFIDIX under the BSD 3-Clause License [Ope99] to the open source community should eventually allow a bigger audience to work with a uniform benchmarking tool [KG06b].

To the best of our knowledge, the only available tool for generically benchmarking arbitrary Java code is the open source JBench [JBe01]. After a short analysis we decided to start PERFIDIX from scratch and not to extend JBench due to its old and unmaintained code base, the class- instead of method-level benchmarking granularity, the need for better statistical output, and the inflexible configuration through property files.

JUnitPerf [JUn01] is an open source extension to the unit-testing framework JUnit [JUn00]. JUnitPerf can be applied to any unit test to perform accurate time measurements. Still, it does not comply with our requirements because it just assures that a unit test fails if it exceeds a given time frame. Multiple runs, statistical output, or support in iteratively improving the performance is not in the scope of JUnitPerf.

In contrast to the generic benchmarking tools, there exist a variety of domain-specific benchmarking tools. All of these focus on a very specific set of functions and present the results in various formats, ranging from unstructured console-based output to full-featured charts. JPerf [JPe07] for example is an open source Java port of iperf [IPe99] which measures IP bandwidth using UDP or TCP, amongst other network-related parameters. JBenchmark [JBe03] is a free benchmark suite for mobile 3D APIs and other Java ME-related technologies. Poleposition [Pol05] is an open source benchmark test suite to compare database engines and object-relational mapping technologies. Most notably, all of these domain-specific benchmarking tools re-implemented the core benchmarking functionality to execute a piece of code multiple times while measuring its execution time and other relevant parameters. Furthermore, each tool has its own way to issue the results or even lacks proper statistics.

Benchmarking is closely related to profiling. Therefore we also looked at Java profilers. Profilers are powerful means to find bottlenecks and optimize existing code. They not only allow analyzing CPU time but also memory consumption, thread execution behavior and plenty of other parameters. However, profilers are complex software requiring quite some skills to run and interpret them. Another disadvantage is that a profiler will not automatically collect or expose statistical evaluations as they appear when running a piece of code multiple times. TPTP [TPT05] is an open source example for a good profiler, JProfiler [JPr01] a commercial one.

Implementation

PERFIDIX 1.0 is a preliminary release our research group is currently working with. It provides the core functionality for generic benchmarking but without the alluded convenience features, i.e., Java annotations and Eclipse integration.

```
public class Example extends Benchmarkable {

  // Setup method called before each method run.
  public void setUp() { ... }

  // Cleanup method called after each method run.
  public void tearDown() { ... }

  // Method to benchmark.
  public void benchMethod() { ... }

}
```

Figure 4.19: PERFIDIX *1.0 example code*

Figure 4.19 shows a self-describing excerpt of the `Example` benchmark. We intentionally designed PERFIDIX 1.0 in the style of JUnit 3.x to foster its adoption by our developers. The Java class `Example` can either be run from Ant using our PERFIDIX 1.0 Ant task or manually by implementing the `main()` method. In both the Ant task and the `main()` method, the developer can configure the number of runs on a per-method and per-class basis. PERFIDIX 1.0 basically measures the execution times in either milli- or nanoseconds. If the developer wants to measure other events, e.g., the number of cache misses of a caching algorithm, he can do so via a customized meter. The customized meter has to be incremented manually whenever a specific event is registered and the result appears along with the default timing measurements in the statistical output.

The human-readable console output of the `Example` benchmark is shown in Table 4.10. Other formats, such as XML or GNUPlot [GNU86], are also available. PERFIDIX 1.0 automatically provides statistical output by calculating the minimum, maximum, average, sum, standard deviation, and 95% confidence intervals for the execution times and other customizable meters. The statistics are calculated on a per-method and per-class basis. The XML output can be configured to contain the results of each run together with relevant metadata, e.g., a time stamp.

Title	Unit	Sum	Min	Max	Avg	Stddev	Conf95	Runs
Example	ms	438	0	3	0.44	0.50	[0.41, 0.47]	10

Table 4.10: PERFIDIX *example output including title, unit, sum, minimum, maximum, average, standard deviation, 95% confidence intervals, and number of runs*

Concluions and Outlook

While PERFIDIX 1.0 proved itself as a generic benchmarking tool, it still lacks important convenience features such as Java annotations and Eclipse integration. Especially the requirement to extend the class `Benchmarkable` and to configure it manually by implementing the `main()` method or run an external tool such as Ant heralded the brainstorming for PERFIDIX 2.0. While the new features still have to be implemented, the design is already stable.

The most notable change will be the use of Java annotations as they are used with JUnit 4.x. Figure 4.20 shows how the `Example` benchmark code excerpt will look like

with PERFIDIX 2.0. The annotations do no longer require the methods to follow a fixed naming, are more expressive, and even will allow configuring the benchmark on a per-method or per-class basis right in the code. With the annotations, a developer could use the same class as a unit test and a benchmark.

```
@BenchClass(runs=1000) public class Example {

  // Setup method called before each method run.
  @BeforeBench
  public void setUp() { ... }

  // Cleanup method called after each method run.
  @AfterBench
  public void tearDown() { ... }

  // Method to benchmark.
  @Bench
  public void benchMethod() { ... }

}
```

Figure 4.20: PERFIDIX *2.0 example code*

Only the tight integration with an integrated development environment – we chose Eclipse because it is widely used and open source – will bring the desired convenience. Therefore, PERFIDIX 2.0 will come along with an Eclipse plugin that allows the developer to right-click on any package or class to find an entry in the context menu similar to JUnit, i.e., "Run as >PERFIDIX Benchmark". The plugin will then run all classes containing `@Bench` annotations with the class-local configuration. An Eclipse view will display the benchmarking progress to give an immediate feedback. This is especially useful for long-running benchmarks and to produce intermediate results. The configuration of each benchmark can be customized through the Eclipse run configuration.

PERFIDIX is a tool for researchers and developers to conveniently and consistently benchmark Java code. It can quickly answer the question which implementation is faster without the need to repeatedly launch a full-fledged profiling application or rewrite the same Java code to measure the execution time again and again. Even though PERFIDIX is simple to use, it still provides a sound statistical output for taking and documenting a decision.

Our first experiences with PERFIDIX show that researchers and developers quickly integrate it in their standard tool set for every-day use because it facilitates the research and development process and supports decision making as well as persistent documentation. Still, successfully benchmarking a piece of code requires quite some artistry [Jai91]. PERFIDIX users must as well carefully consider several aspects and find a common sense of "do's and dont's". The JVM for example with its garbage collector, heap allocation internals, threads, object pools, and buffers, as well as the environment outside the JVM can influence the benchmark results in manifold ways.

A general advice for the developer is to first profile the whole application to find its runtime-bottlenecks. Having identified the pièce de résistance one can iteratively trim the equivocal code and document the progress with PERFIDIX. A final run

with the profiler should then reflect the improvements.

Initially developed as a simple benchmarking framework to avoid error-prone repetitive manual tasks, PERFIDIX will be integrated into a development environment as well as enriched with a chart generator. The achieved progress can then constantly be tracked while the code is being optimized or new concepts and ideas are introduced.

4.3.3 VISIDEFIX

Introduction

Recent block-based native XML storage systems touch blocks according to the XQuery engine's execution plan. The resulting access patterns are virtually unknown and potentially cause many expensive disk seeks. Visualization comes to the rescue when extensive log files must be analyzed – a tedious and difficult task. The dynamic time-based block-touch animation as well as the static block-type information of VISIDEFIX foster the insight into the performance-critical internals of the XML storage and help to optimize both the block layout and the XQuery engine to speed up queries.

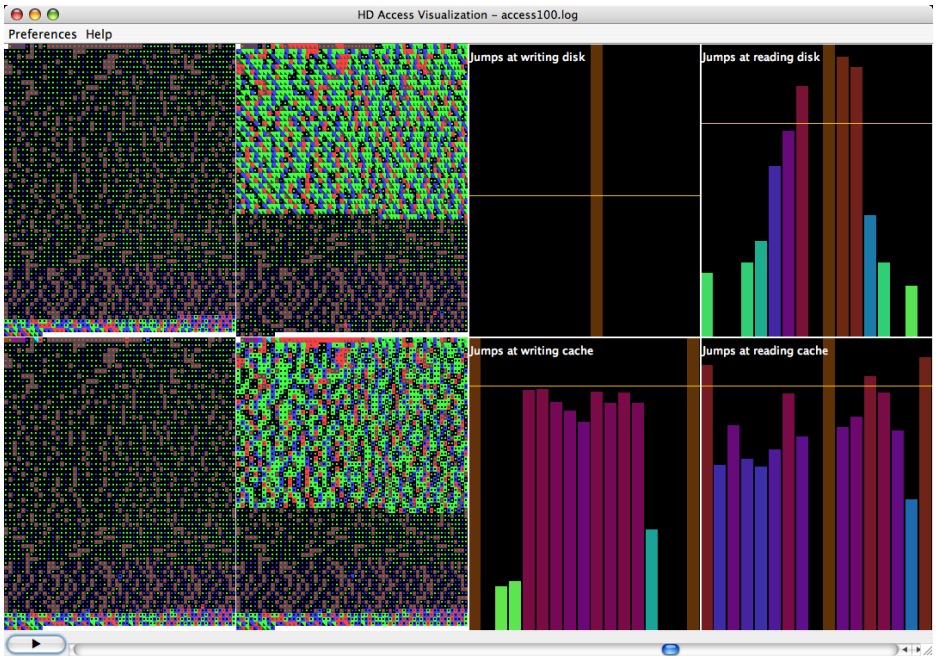


Figure 4.21: VISIDEFIX *is a tool to explore block access patterns of the native XML storage* TreeTank. *The tool consists of four block access diagrams (read disk, write disk, read cache, write cache) and four jump distance histograms. The detailed view of the disk read operations shows touches of both metadata and data blocks (gray vs. black background) by highlighting them over time. The jump histogram (top right) reveals that the storage is already fairly optimized for the shown query as the distribution of disk read operations is focused around the center, which indicates that data is read mostly sequentially, avoiding expensive seek operations*

Efficient storage and retrieval of XML data is a challenging research area. TreeTank competes in this field by storing native XML data in blocks of a random-access

device. The XML data is organized in three main data structures, i.e., the Name Map, the Node List, and the Value Map. The node list stores the tree structure of the XML data. Each element, attribute, or text node appears in the node list in `pre` order which corresponds to the `pre` order (i. e. first-depth) traversal of the XML tree [Gru02]. A prefix tree locates each entry in the node list by its `pre` position. Tag and attribute names as well as attribute and text values are stored in the name and value map respectively. Both names and values are accessible through so-called tries. All tries are stored in metadata blocks while the node list, name, and value map entries are stored in data blocks.

During the evaluation of the XMark benchmark [SWK+02] we run different queries against TreeTank. The limiting performance factor, random disk I/O, clearly asked for an optimized layout of the blocks to efficiently support different access patterns. The analysis of the block layout as well as the access patterns is effectively supported by the visualization tool described in the next section.

Implementation

VISIDEFIX is a visualization tool that supports the analysis of block layout and access patterns of the native XML storage TreeTank (see Figure 4.21). The basic idea is to see where data and metadata are stored on the disk to evaluate both the locality of block storage as well as the correctness of block access of XML queries. Furthermore, our tool animates the execution of queries by visualizing block access operations on disk and cache to efficiently support the system engineer in his task to verify hypotheses about the system's behavior as well as to formulate novel hypotheses and search for their causes.

The visualization is designed in a way that each storage block is represented through a small rectangular icon on the screen. Color depicts the type of block as illustrated in Figure 4.22. We distinguish between three states of a block (*untouched, recently touched,* and *touched*), which are represented through distinct icons. Note that there are variations of the icon for *recently touched*; the more a block is touched, the more intense is the color of the bottom left corner of the icon. Over time, icons of *recently touched* blocks continually fade out until they reach the state of *touched*. The strong contrast between "untouched icons" and "touched icons" enables us to easily distinguish between used and unused areas of disk and cache during query execution. Conceptually, we classify the blocks into two groups: data (i.e., XML nodes and their values) and metadata (i.e., root and trie) blocks. Black (gray) is employed as background for the data (metadata) icons.

Current random-access devices with block-oriented interfaces provide a logical block order from the 0th to the nth block. We opted for a line-by-line arrangement of the blocks in our block access diagrams, alternating in forward and backward direction to better preserve visual clusters of subsequent blocks. It is possible to retrieve details (i.e., block number and number of touches) about each block by clicking on it within the diagram. The actual arrangement of the icons is based on a recursive pattern [KKA95] implementation. We abandoned more complicated parameter settings of the recursive pattern due to the cognitive overhead required for proper interpretation of the patterns.

The arrangement of data and metadata blocks already gives a feeling for the effectiveness of the used allocation scheme. In Figure 4.23, the upper part is dominated by value blocks (green) with relatively few trie blocks (red), which store metadata of the index structure, whereas the middle part shows proportionally more trie and

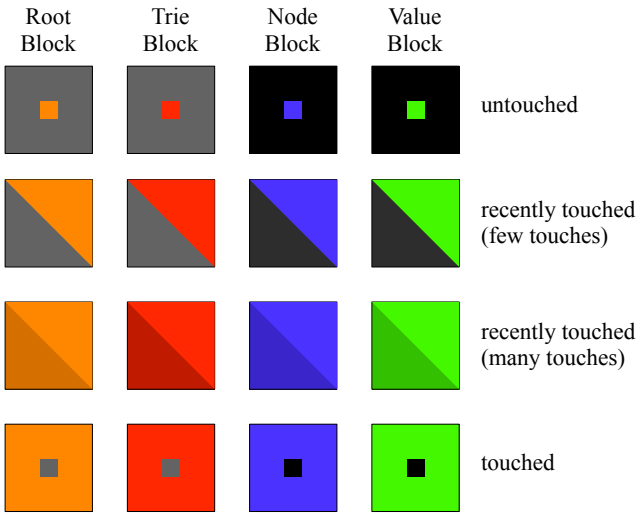


Figure 4.22: *Icons for metadata (gray) and data blocks (black). Variations of the base icon for each block type are utilized to show different usage states*

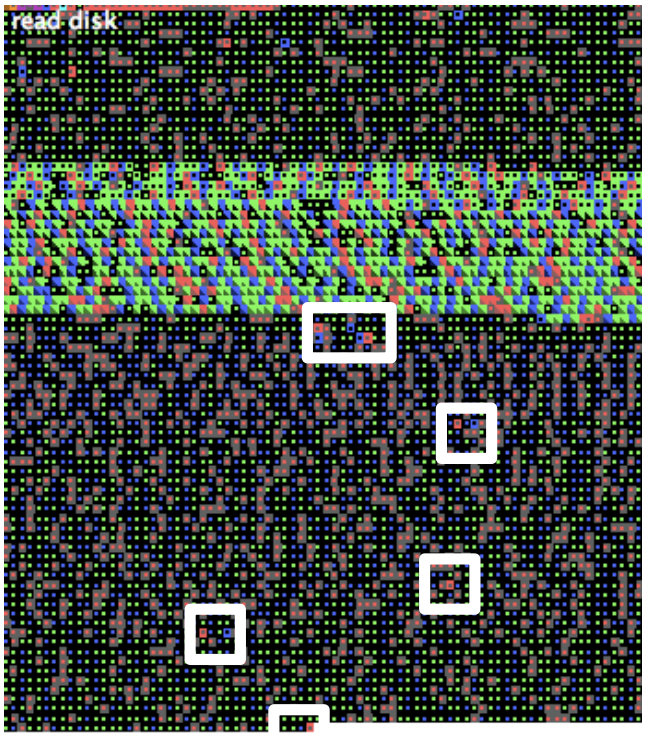


Figure 4.23: *Outlier blocks touched in the course of a query are directly visible and reveal valuable insight to the system engineer for further optimization*

node blocks (blue). Whether this partitioning is good or bad depends largely on the characteristics of queries for which the storage should be optimized.

In addition to the block access diagrams, VISIDEFIX offers linearly and logarithmically scaled histograms that are used to display counts of block jump distances (see Figure 4.21). The value ranges of the histogram bins increase from the middle bin to the outer bins. We implemented a normal mode to investigate jumps for short time spans as well as a cumulative mode enabling analysis of larger time spans. The horizontal bar denotes the average bin size for each histogram.

Conclusions and Outlook

The visualization of block layout and access patterns of the native XML storage TreeTank not only endows us with an excellent tool to analyze different queries, but also gives many valuable hints how to organize blocks more efficiently for various workloads. One benefit of the visualization has been the identification of a query eagerly touching unnecessary blocks.

VISIDEFIX proofs to be very instructive for students to understand a block-based native XML storage system. Future work will include visualization of the internal block fragmentation due to updates, a faster visualization that allows for online observation of TreeTank as well as interaction possibilities to enlarge regions of interest.

4.4 Summary

We implemented two versions of our concept TreeTank to evaluate and affirm our findings. First, we show that the underlying concepts definitively allow for linear scalability while maintaining logarithmic update characteristics. This is most important, because we could not have continued without this result. Second, we show that, while keeping the linear scalability, we can move on to node-level granularity. This opens the door to switch to the *evolutionary* approach with its fine-granular modification history.

We could not have done our work on TreeTank without jSCSI, PERFIDIX, and VISIDEFIX. All three tools not only proved themselves as valuable time savers, but they also showed that:

- a Java-based iSCSI implementation provides block access performance on par or better than a native C-based iSCSI implementation while providing scalable block-level access to a single device or a whole pool of devices to any Java-based native XML storage or any other application with the requirement to access block devices.
- a Java benchmarking tools allows to quickly compare and evaluate different algorithms and implementations while providing publication-ready benchmarking output and save a lot of time of the researcher.
- a realtime block access monitoring tool greatly assists the researcher in finding bottlenecks and comparing different algorithms and implementations of native XML storage applications.

Chapter 5

Interfaces

The eXtensible Markup Language (XML) is more than a unified data exchange and storage format. We suggest the exploitation of XML and look at it as a fine-granular node tree, which is grown up through a sequence of user modifications. The Representational State Transfer (REST) is the perfect candidate to expose XML resources as well as their full version and modification history to the World Wide Web. Extending the idea of XML and REST along the natural modification-driven temporal dimension breeds something, which is scalable, robust, simple, and yet extensible enough to effectively enrich striving applications such as personal information management, collaborative document authoring, distributed content management, or Geographic Visual Analytics. In this chapter, we introduce Temporal REST, i.e., an interface and protocol to access web-based XML resources as well as their full version and modification history. We describe the underlying data model and show how it solves problems inherently arising from temporal interactions in a pragmatic and straightforward way. In addition, we provide a case study to demonstrate the power of Temporal REST due to its elegance and true simplicity. Finally, we motivate future work including the implementation of back-end services as well as front-end applications – both of which will mutually benefit from Temporal REST.

5.1 Principles

5.1.1 The Importance of REST and XML

Twelve years after the introduction of HTTP, Roy Fielding coined the word REST [Fie00]. REST is a set of network architecture principles, which outline how resources are defined and addressed. Practically speaking, REST defines a simple and scalable interface to exchange resources over HTTP. Each resource must be uniquely addressable through hypermedia links meeting a universal syntax. A well-defined and typically small set of HTTP operations specifies how to proceed with the obtained resource. The basic operations are POST to create a resource, GET to read a resource, PUT to update a resource, and DELETE to remove a resource. RESTful web services have appeared all over the Internet and compete with already-established protocols. The simplicity and elegance of REST makes alternatives such as the XML-based SOAP, binary CORBA [Gro04], or DCOM [Net96] look like unhandy fellows. Web application frameworks such as Ruby on Rails [Han03] quickly

adopted and favored REST. Virtually any programming language or framework nowadays has tools, e.g., Restlet for Java [Con05] or Astoria for .NET [Mic07], to facilitate RESTful application development. However, Roy Fielding did not provide a detailed description on how to use REST for a specific application. It is left to the developer of each application to specify how exactly the interface should look like and how the resources should be accessed.

In the wake of the unprecedented growth of the Internet, the need for a unified resource-encoding format culminated in the standardization of XML. Since then, XML has started to conquer the world as a universal data exchange and storage format. The human-readability of XML along with its rich toolset consisting of XPath, XSLT, XQuery, among others, lead to a quick adoption of XML for protocols such as SOAP, which allows to access web-based objects, BPEL [IBM07], which allows the modeling of high-level business logic, or Atom [NS05], which is a protocol to feed news. Even the shady side of XML, i.e., its sheer verbosity and excessive demand for processing power could not really impair its success. Rather, more and more traditional relational database systems such as IBM DB2 [IBM70], Oracle Database [Ora79], or Microsoft SQL Server [Mic89] have started to natively store XML data types for improved performance and interoperability. Other database systems, e.g., X-Hive [XH05a], no longer support the traditional relational model but focus on native XML storage. In contrast to traditional (object-) relational databases, XML has a convenient feature: It supports a data-before-schema approach, which does not require the specification of a schema before the storage of any data. Finally, the Efficient XML Interchange Working Group [W3C08] has a strong intention to speed-up the XML processing to reduce its size through a binary encoding.

5.1.2 A Temporal Extension to REST

While there exists a variety of solutions to access XML resources over the Web, there is – to our knowledge – no generic and unified solution to conveniently access all of:

1. The **current version** of the XML resource or any subset thereof.
2. The **full version history** of the XML resource or any subset thereof.
3. The **full modification history** of the XML resource or any subset thereof.

Our approach exploits XML by tightly integrating it with REST. We want to put aside the antiquated view of XML as a simple data exchange and storage format and discover what it really is: a fine-grained tree of nodes, which evolves over time through user modifications. If we let ourselves to view XML as a growing tree of nodes, we realize that we can access single nodes or whole sub-trees, i.e., XML fragments, within a temporal dimension in a unified, scalable and robust way.

We want to query the XML the way it was stored at any past point in time. Note that a point-in-time references a version, i.e., a state of some resource. In addition and in stark contrast to all widely used interfaces and protocols, we want to randomly query for user modifications between any two past points in time. Only if we consider the whole life cycle of an XML resource including the past versions and the (transaction-based) modification history, we will get a complete idea of its true power. We suggest Temporal REST as an interface with its related protocol message exchanges to generically implement our idea to exploit web-based XML resources.

According to the Pareto principle [PMA40], our proposal is simple enough for the average web application developer and at the same time it is extensible enough to be used with complex setups.

To get a better idea about the intention, imagine a Temporal REST web service providing access to the corporate-wide XML-encoded contacts in a temporal fashion. In this example, the CEO contact is referenced by ID 3, version 7 of the CEO contact has a time stamp equal to `20000624T1400`, and version 14 of the CEO contact has a time stamp equal to `20001231T1400`.

Among others, the following use cases are possible:

1. Retrieve all current contacts:
   `GET http://.../contacts`
2. Retrieve the current contact of the CEO rooted at node ID 3:
   `GET http:/.../contacts/3`
3. Retrieve all contacts at version 7:
   `GET http://.../contacts/(7)`
   or
   `GET http://.../contacts/(20000624T1400)`
4. Retrieve the contact of the CEO at version 7:
   `GET http://.../contacts/(7)/3`
   or
   `GET http://.../contacts/(20000624T1400)/3`
5. Retrieve all modifications applied to the contact of the CEO between version 7 and 14:
   `GET http://.../contacts/(7-14)/3`
   or
   `GET http://.../contacts/(20000624T1400-20001231T1400)/3`

5.1.3 The Current State is Not Sufficient

In addition to the related work presented in Chapter 2, we identified three categories of related work. First, the systems without any temporal support. Second, the systems with support for access to past versions. Third, the systems with support for access to past versions and some kind of modification observation. Almost all current file and database systems belong to the first category. Note that modern file and database systems actually do perform some kind of journaling or transaction logging to support crash recovery or transactional behavior but they only use it for internal purposes and do not provide a public interface.

The second category contains an increasing number of systems. However, all of these systems mainly suffer from the fact that the modification history has to be extracted on the application layer by comparing two different versions. This extraction is based on expensive (binary) delta calculations such as Xdelta [Mac08]. In addition, the deltas do not immediately reflect modifications on fine-granular tree-based data structures such as XML. Concurrent versions systems such as CVS [Gru86], Mercurial [Mac06], and WebDAV-based [GWF+99] Delta-V [CAE+02] serve as good examples for the second category. All three systems provide access to the current and all past versions of XML (and other) resources but they do not work at the fine

granularity of XML and they do not provide an interface to query the modification history of the XML at its natural granularity, i.e., the node or sub-tree level. In the file system world, Hammer [Dil08] and ZFS [BM04] are recent additions. While Hammer supports access to all past versions, ZFS only keeps a subset of the past versions, i.e., user-demanded snapshots.

The third category is the youngest and smallest one. Systems in the third category allow to stream modification events to other systems for backup or other purposes. They do not aim at providing random access to any past modification as we do with Temporal REST. Apple has only recently started to make the local file- and directory-level modifications visible to the applications through FSEvents [Ars07a]. Apple's Time Machine [App07] is an excellent example application to consume these events to perform an incremental backup. Note that FSEvents does only remember that a given file or directory changed. It does not remember the fine-granular changes of, say, the XML node tree stored within a file. The Content Repository API for Java Technology Specification [Pro07] comes closest to our idea as it optionally supports the concepts of versioning, activities, and observation. Versioning works similar to concurrent versions systems. Activities group modifications and make them accessible for later re-use. Observation encapsulates modifications into events to propagate them to all interested parties. However, all three concepts are kept separately and optional – a fact which complicates the every-day use.

5.2 Data Model

5.2.1 Session- and Transaction-Based Access

Our data model encapsulates each web-based XML resource within a single session, i.e., each session is responsible to coordinate the access to a single XML document together with its related version and modification history. A session allows multiple concurrent read and one single write transaction at any time. Read transactions can access all past modifications and versions of the XML document up to the last successfully committed version. A write transaction creates a new version of the XML document upon commit or drops all changes upon abort. Starting with one, each version is assigned a positive number in increasing order. In addition, each version is at least tagged with a time stamp, an author, and a commit comment. Supplementary meta-data can be added as required. While a read transaction only allows select operations, the write transaction additionally allows insert, update, and delete operations. A read transaction can sequentially execute multiple select operations. The write transaction can sequentially execute any operation until the write transaction is either committed or aborted. The isolation is clearly given with this model, i.e., only the single write transaction will see dirty data. The response of an operation always consists of a sequence of items as described with the data model of XPath 2.0 and XQuery 1.0. The relationship of the involved entities is shown in Figure 5.1.

For our initial version of Temporal REST, we have chosen to only support a single write transaction for each session at any time. While this might look too restrictive or cause bottlenecks due to the serialization of concurrently issued write operations, we favor simplicity from both the usability and implementation perspective. Other widely deployed systems such as the full-text framework Lucene [Apa97] and the file system ZFS also support only a single write transaction and they are still perfectly useful for real-world use-cases. If the serialization of multiple concurrently issued

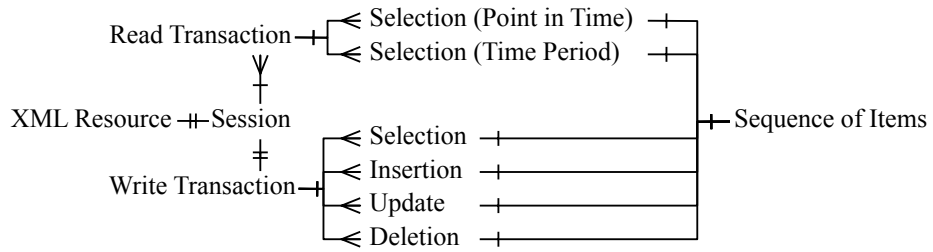


Figure 5.1: *Temporal REST data model. The relationships are depicted according to Barker's notation [Bar90]*

write transactions still causes a bottleneck, a XML resource can be split up into smaller ones, i.e., to allow more independent write transactions. Finally, an application can implement a sophisticated access and locking model, which includes two or three phase commits spanning multiple sessions. As undo and redo operations are inherently supported by Temporal REST (the past versions are always available), an already committed version can quickly be reverted to a past one.

Security with respect to confidentiality and integrity is cared for – if required – with the transport layer security protocol on top of which HTTP works. Authentication and authorization are handled with the readily available mechanisms of HTTP. A clear separation of concerns tremendously facilitates the specification and implementation of Temporal REST.

5.2.2 XML Fragment Identification

There are two fundamental ways to access nodes and sub-trees, i.e., XML fragments, within an XML resource. First, the traditional axis-navigation or query-based access. Second, the ID-based random access. Temporal REST supports both and complements them with a temporal expression as described later.

XML IDs enable the user to tag the XML document and to quickly access the XML fragment by providing this XML ID. However, most XML nodes are not tagged with such a XML ID and remain inaccessible from the XML ID perspective. We suggest the tagging of at least all element nodes with a system-generated REST ID. Text nodes or attributes are accessible through their parent node. Other XML nodes such as comments or processing instructions may be tagged by the system on demand. One advantage of having the system to do the REST ID assignment is that the REST ID remains stable throughout versions and modifications, i.e., a node or its modifications can be accessed irrespective of the version or position in the tree. Another advantage is the guarantee of the existence of an ID. The system can make the REST IDs visible by tagging the serialized XML with REST ID attributes bound to the namespace of Temporal REST. Consequently, the user may choose to use this information to quickly access the nodes later on in an ID-based random-access fashion.

Each insertion operation assigns unique immutable REST IDs to all new element nodes. This assignment is made by the back-end that stores the XML and does not affect any existing user-assigned XML ID. REST IDs are numerical and they are incrementally assigned starting at one. The document root has the REST ID one. REST IDs do not necessarily need to be assigned in document order and they must not change once assigned to a node. In addition, we suggest not reusing REST IDs.

This reduces the confusion due to reassignments in future versions. Since deletions are less frequent than insertions with most real-world workloads [BR01], the loss of number space is considered to be negligible. Figure 5.2 shows the assignment of REST IDs.

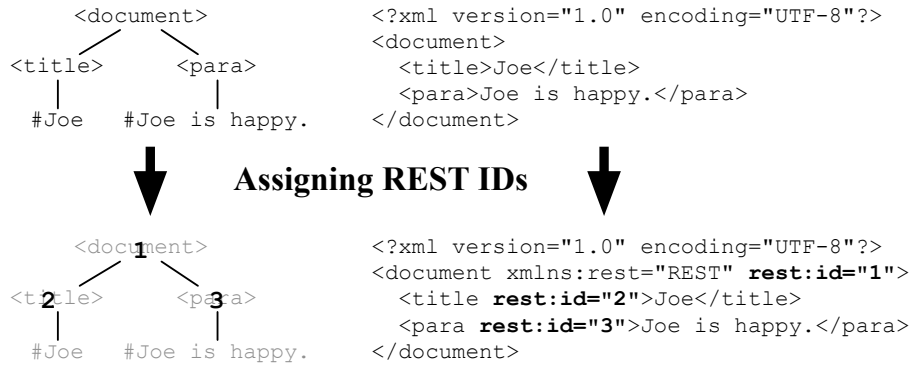


Figure 5.2: *Assignment of REST IDs. Any XML fragment or document can be depicted as an unranked ordered tree. The REST ID makes sure that every element node gets its own unique immutable identifier. '<' and '>' denote element nodes and '#' denotes text nodes in the node tree. A simple XML resource storing a document serves as an example*

5.2.3 XML Fragment Modification

Each insertion, update, or deletion of a XML node results in a modification event. Each write transaction commit groups the modification events into one version and assigns a time stamp, an author, and a comment to the whole version. Temporal REST communicates modifications by encapsulating the modified node within an item element. The item element contains the REST ID of the modified node as well as version, time stamp, author, and comment information. As such, both the insertion and the deletion can be considered as setting a node to a new value. Deletion sets the node to the empty node.

We opted for this approach for two reasons. First, we can streamline the transport of XML fragments and modifications within the original XPath 2.0 and XQuery 1.0 data model, i.e., within a sequence of items. Second, the back-end can combine the storage of the modification event and the result of the modification. Section 5.4 will show, how this is achieved in practice.

Read transactions can select XML fragments in any version. In addition, read transactions can select the XML fragment modifications, which took place between any two versions. Since the version number may not be convenient enough, the system must support the resolution of a time stamp into the closest version number.

5.2.4 XML Fragment Serialization

The default behavior of the original XML data model is to serialize the whole sub-tree of a XML node returned as an item. From both the practical and safety perspective, it may be reasonable not always to return the whole sub-tree but to limit its global depth and the per-node fan-out. Whenever a serialization truncates a XML fragment due to depth or fan-out limitations, it will tag the last serialized node with the depth or fan-out limitation to inform the user and allow to retrieve the

missing nodes with a consecutive request. In addition, every element node is tagged with its REST ID. If the sequence itself contains too many items, the sequence can be paged.

5.3 Operations

5.3.1 Select

The select operation allows the retrieval of a sequence of items as defined with the XPath 2.0 and XQuery 1.0 data model. Each item is an atomic value, a XML node, or now also a modification event. The selection can be query-based, i.e., an XPath 2.0 or XQuery 1.0 expression, or REST ID-based. Temporal REST will restrict the execution domain of both the query and the REST ID according to the temporal expression either selecting a point in time or a time period (see Figure 5.3). While a query may return a sequence of multiple items, an access solely based on a REST ID will return a sequence with at most one item. If the query and REST ID approach are combined together, the query treats the node with the given REST ID as the root node of the query. The query-based approach allows to add new query languages in the future and to express complex queries including operations such as full-text search or joins. The REST ID-based approach allows to directly select an item with optimal performance since the system does not have to compile and optimize the query.

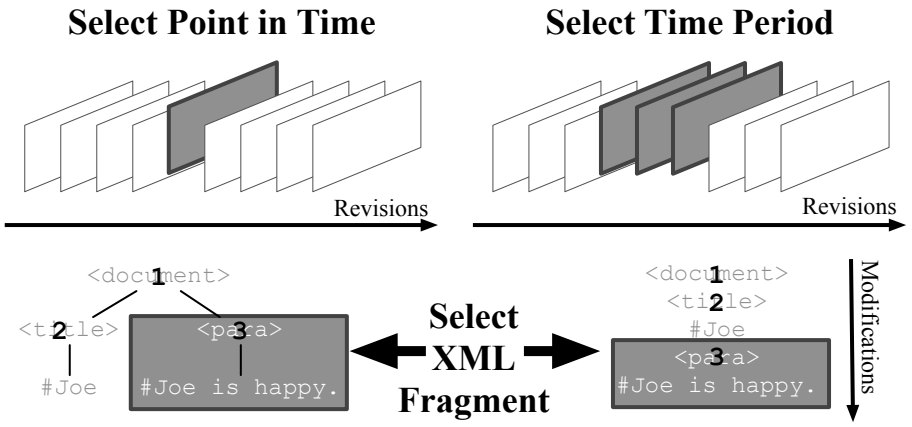


Figure 5.3: *Selection. The left side shows the selection of an XML fragment, as it was stored at a given point in time. The right side shows the selection of the modifications on a XML fragment during a given time period*

The temporal expression must be enclosed with round brackets '(' and ')' and contain a single point in time or a time period consisting of two points in time separated by a dash '-'. A point in time can be a version number, an ISO date in short notation, i.e., without dashes or colons, or nothing, i.e., the last successfully committed version. A single point in time will retrieve the XML fragments as they looked like at the given version. The time period will retrieve the modifications between (and including) the two provided points in time in the according order. Leaving away the temporal expression automatically causes a fallback to the last successfully committed version for backwards compatibility.

5.3.2 Insert

A single node or a whole sub-tree can be inserted either as the first child of an existing node or as its right sibling. As such, the insert operation requires a query selecting a number of nodes or a REST ID besides the actual XML fragment to insert. During the insertion process, the back-end system will assign the REST IDs as described above. Note that the insertion of an attribute must be made with the PUT operation changing the whole node.

5.3.3 Update

A single node can be replaced with or without the replacement of its sub-tree. Again the updating operation requires a query selecting a number of nodes to update or a REST ID besides the actual XML fragment to represent the updated node or sub-tree. Restricting the effect of the update to the node (not effecting its sub-tree), allows the insertion of an attribute into an existing node without changing its whole sub-tree.

5.3.4 Delete

Whenever a node is deleted, the node and its sub-tree are purged from the system (but not from the past versions). The deletion operation requires a query or a REST ID to select the nodes to delete. Note that we do not suggest to recycle REST IDs to avoid confusion when once-deleted REST IDs reappear in later versions, especially, when they appear for different new nodes.

5.4 Case Study

Collaborative document authoring serves as a perfect case study. Let us assume a workflow that specifies the role of the person, the activity, and the exact time this activity has to be performed during the publication process. Having different stages, the workflow involves multiple people who take on different roles such as author or reviewer who perform tasks sequentially or concurrently. If the underlying document is stored as XML, e.g., in OpenDocument [All02] or DocBook [OAS91] format, then the application layer can conveniently provide temporal functionality. At any time, the author or reviewer can effortlessly observe who has done what since the author or reviewer last looked at the document. The Temporal REST interface allows to quickly visualize the modification history or to swiftly create individual Atom news feeds for the involved people, i.e., to transform the response with XSLT into valid Atom XML. While the application still has to model and implement the workflow – a task, which is an art of its own – it is greatly simplified because it does not have to consider the design, interface, message exchange, and implementation of a specific temporal functionality: it can solely rely on Temporal REST.

The document we are working on will see a sequence of modifications as described with Table 5.1. The HTTP request and response pairs to perform these modifications are listed in Table 5.2 (Rows 1 to 3) alongside with a query selecting a point in time (Row 4) and a query selecting a time period (Row 5).

	User Intention	Required Modifications	Resulting Revision

1	Add title 'Joe' and paragraph 'Joe is happy.' to document	REST ID 1:
		Insert `<document>` as first child of REST ID 0
		REST ID 2:
		Insert `<title>Joe</title>` as first child of REST ID 1
		REST ID 3:
		Insert `<para>Joe is happy.</para>` as right sibling of REST ID 2
			`<?xml version="1.0"?>`
			`<document>`
			`  <title>Joe</title>`
			`  <para>`
			`  Joe is happy.`
			`  </para>`
			`</document>`

2	Rewrite paragraph to 'Mike is happy.'	Update REST ID 3 to
		`<para>Mike is happy.</para>`
			`<?xml version="1.0"?>`
			`<document>`
			`  <title>Joe</title>`
			`  <para>`
			`  Mike is happy.`
			`  </para>`
			`</document>`

3	Remove title	Delete REST ID 2	`<?xml version="1.0"?>`
			`<document>`
			`  <para>`
			`  Mike is happy.`
			`  </para>`
			`</document>`

Table 5.1: *Example sequence of user modifications*

Row 1 of Table 5.2 shows the initial import of an XML document into the repository of XML resources. As a reaction to this HTTP POST request, the server-side session initiates a write transaction, inserts the XML fragment given in the request body, tags all inserted element nodes with REST IDs, commits if no error was encountered, and responds with a sequence bound to the committed version, i.e., 1, and containing a single item, i.e., the inserted XML fragment.

Row 2 of Table 5.2 replaces the XML fragment rooted at node with REST ID 3. Again, the server-side session initiates a write transaction, overwrites the existing XML fragment with the XML fragment of the request body, tags all new element nodes with REST IDs, commits if no error was encountered, and responds with a sequence bound to the committed version, i.e., 2, and containing a single item, i.e., the updated XML fragment.

Row 3 of Table 5.2 removes the XML fragment rooted at node 2. The server-side session initiates a write transaction, removes the requested XML fragment, commits if no error was encountered, and responds with a sequence bound to the committed version, i.e., 3, and containing a single empty item to mark the deletion. Note how the REST ID propagates to the item node because the item does not contain any node anymore.

Row 4 of Table 5.2 shows a query for a given point in time, i.e., version 1. Here, we show an XPath 2.0 expression restricting the result to a sequence of items, each containing the text of a paragraph node. The server-side session initiates a read transaction bound to the given version, compiles and executes the XPath 2.0 expression, and returns the result.

Row 5 of Table 5.2 shows a query for a time period, i.e., all modifications which took place between revision 2 and 3 (inclusive). The server-side session initiates a read transaction bound to the newer revision and retrieves all modifications between the newer revision and the older revision.

	HTTP Request	HTTP Response
1	POST http://../document <?xml version="1.0"?> <document> <title>Joe</title> <para> Joe is happy. </para> </document>	<?xml version="1.0"?> <rest:response xmlns:rest="REST"> <rest:sequence rest:revision="1"> <rest:item> <document rest:id="1"> <title rest:id="2">Joe</title> <para rest:id="3"> Joe is happy. </para> </document> </rest:item> </rest:sequence> </rest:response>
2	PUT http://../document/3 <para> Mike is happy. </para>	<?xml version="1.0"?> <rest:response xmlns:rest="REST"> <rest:sequence rest:revision="2"> <rest:item> <para rest:id="3"> Mike is happy. </para> </rest:item> </rest:sequence> </rest:response>
3	DELETE http://../document/2	<?xml version="1.0"?> <rest:response xmlns:rest="REST"> <rest:sequence rest:revision="3"> <rest:item rest:id="2"/> </rest:sequence> </rest:response>
4	GET http://../document/(1)? //para/text()	<?xml version="1.0"?> <rest:response xmlns:rest="REST"> <rest:sequence rest:revision="1"> <rest:item> Joe is happy. </rest:item> </rest:sequence> </rest:response>
5	GET http://../document/(2-3)	<?xml version="1.0"?> <rest:response xmlns:rest="REST"> <rest:sequence> <rest:item rest:revision="2"> <para rest:id="3"> Mike is happy. </para> </rest:item> <rest:item rest:revision="3" rest:id="2"/> </rest:item> </rest:sequence> </rest:response>

Table 5.2: *Example sequence of HTTP request and response pairs for our five main use cases. The POST and PUT HTTP requests in rows 1 and 2 show the HTTP XML fragments passed in the HTTP body. The HTTP requests in rows 3, 4, and 5 do not have a HTTP body. Note that XPath 2.0 expressions must be URL-encoded – a fact which may lead to a loss of expressiveness of XPath 2.0. Also note that the resulting two modification items in row 5 are similar to the two resulting state items in rows 2 and 4 with the only exception that resulting modification items also contain the revision number. Comparable to row 4, row 5 could also restrict the resulting modification fragments with a XPath 2.0 expression passed in the requesting URL*

While we have good experiences with the main use cases, there are many others open for discussion. First of all, there are many variations on how to express a modification. E.g., an insertion of a XML fragment as the first child of some node can be expressed as a change of this (parent) node. Second, the complexity increases if not only element but also text nodes, attributes, or other nodes are tagged with a REST ID. This is not due to the load on the system, which does not change, but mainly due to the fact that text and attribute nodes must be surrounded with auxiliary metadata element nodes if modified directly and if they are not contained within a surrounding element node.

We experimented with grouping several HTTP requests into one, i.e., group several operations such as select or insert into one HTTP request. As a consequence, the request URL loses its expressiveness because it cannot transmit any information any more as this might have to be shared with all contained operations. The whole request metadata must be packed into the HTTP body. This makes it necessary to express the session context and the read transaction or write transaction boundaries within the request body. If just a single request is issued, this can be implicitly encoded in with the HTTP command and URL. A clear advantage of request grouping is the fact that several modification operations can be executed within a single transaction. In addition, more metadata can be encoded into the request body than into the request URL. E.g., it is not convenient to encode a complex XQuery expression into the URL.

Besides theoretical reasoning, we implemented a preliminary prototype of Temporal REST to back our estimates about performance and space requirements based on TreeTank. We found that our simple prototype showed performance in the same order of magnitude (approx. 30% overhead) to SAX when retrieving the whole XML resource in any given version. The same holds for XPath 2.0 expression evaluation. As soon as it comes to REST ID-based random access, our prototype clearly out-performed SAX. Note that these performance results are common when one compares any other existing native XML database with SAX. The differences start to show up when the modifications are queried. While this is only possible with our prototype, all others fail due to missing functionality. We can stream the modifications at one half of the performance of streaming a version. In addition, our prototype is able to shrink the first version to about one half of the size of the original XML file. Each write transaction commit then roughly adds a few kB of data, depending on the number of modified nodes. For single node modifications it can be as low as a few hundred bytes. For many nodes, it is roughly one half of the original XML fragment size.

For implementations based on (object-) relational databases, it is important to agree on a generic mapping between XML files and relational tables. This is necessary to guarantee the consistency of the interface irrespective of the underlying back-end implementation, i.e., a native XML or a relational database. As long as the data schema can be mapped to relational tables conforming to the relational normal forms, this is not a problem and actually the case for the vast majority of XML files or relational database schemas.

5.5 Summary

Temporal REST is a new paradigm on how to exploit web-based XML resources. Instead of solely thinking about XML as a unified resource exchange and a storage

format, we promote the idea of looking at XML as a growing tree of nodes. We want to provide a generic and unified solution to conveniently access all of:

1. The **current version** of the XML resource or any subset thereof.
2. The **full version history** of the XML resource or any subset thereof.
3. The **full modification history** of the XML resource or any subset thereof.

We see potential applications, e.g., in the area of personal information management, collaborative document authoring, content management, or geographic visual analytics. The interdisciplinary character emerging from the fact that different sciences and businesses will develop applications on top of Temporal REST makes it especially attractive. All above-mentioned applications currently use XML and some kind of web-based interaction. The major advantage of Temporal REST lies in its expressive and convenient interface vastly reducing the design and implementation complexity formerly faced with each new application. While Temporal REST facilitates the look into the past by technical means, it will remain for every application, their users, national or international law, and, as such, our society to decide when to eventually erase past versions. The trade-off between archiving, usability, and privacy is likely to cause enthralling discussions.

We invite the web application development and services community as well as (object-) relational and native XML database implementers to scrutinize Temporal REST, implement prototypes, and contribute new use cases and practical findings. We strongly believe in the worthiness of our idea will promote our idea towards a Request For Comment. If Web 2.0 is the web for social and collaborative interaction, Web 3.0 may become the temporal web, i.e., a global time machine.

Chapter 6

Applications

It is important to verify our conceptual contributions in real-world applications. We selected the field of Geographic Visual Analytics because it is exemplary for the following applications.

Interactive (Section 6.1) Cartography and computer science powerfully attract each other to band together into a melting pot stimulating research in interdisciplinary fields such as Geographic Visual Analytics. Cartography with its rich history of practice and science brings along visualization knowledge and large data sets. Computer science with its short but vigorous evolution brings along data structures, algorithms and hardware to make the visualizations and large data sets come alive for unprecedented user interaction. However, it turns out that the traditional workflow suffers from noticeable delays and read-only limitations which practically hinder the every-day convenient and fluent interaction with large data sets. We show how to streamline the traditional workflow by eliminating the intermediate data conversion step through switching to a native XML database. We suggest to use a RESTful interface providing scalable temporal read-write access. Finally, we provide preliminary measurements based on our prototype implementation named TreeTank providing both compressed storage and fast SVG delivery.

Collaborative (Section 6.2) We propose a new, streamlined, two-step geographic visual analytics (GVA) workflow for efficient data storage and access based on the native web XML database TreeTank coupled with a Scalable Vector Graphics (SVG) graphical user interface for visualization. This new storage framework promises better scalability with rapidly growing data sets available on the Internet, while also reducing data access and updating delays for collaborative GVA environments. Both improve interactivity and flexibility from an end-user perspective. The proposed framework relies on a REST-based web interface providing scalable and spatio-temporal read-write access to complex spatio-temporal data sets of structured, semi-structured, or unstructured data. The clean separation of client and server at the HTTP web layer assures backward compatibility and better extensibility. We discuss the proposed framework and apply it on a prototype implementation employing world debt data. The excellent compression ratio of SVG as well as its fast delivery to end users are encouraging and suggest important steps have been made towards dynamic, highly interactive, and collaborative geovisual analytics environments.

6.1 Large-Scale Interactive Geographic Visual Analytics

6.1.1 Introduction

Due to a lot of neurons for visual processing, humans are good at visually identifying patterns. In Geographic Visual Analytics, this is used for knowledge discovery in databases (KDD) or exploratory data analysis (EDA). A challenge of current research in this highly interdisciplinary field is to synthesize information and to derive insights from handling massive, dynamic, ambiguous, and often heterogeneous data sources [KMSZ06].

The scientific objective is to understand how both individuals and teams carry out analytical reasoning and decision making tasks with complex information and to use this understanding to develop and assess information and communication technologies for this purpose [MCM+06]. There are different ways to analyze complex information and a number of different activities in science where such applications are useful. Gahegan [Gah07] presented different approaches and types of methods to handle multivariate data. Thanks to continuous development and use of on-demand geo-visualization tools, it should be possible in the future to propose highly adaptable representations to the current needs of users in an inexpensive way [Kra98, Gah07].

To provide windows into the complexity of phenomena and processes within complex and linked data sets, MacEachren and Kraak [MK01] focused research challenges in geo-visualization on four themes: representation, visualisationcomputation integration, interfaces, and cognitive or usability issues. Modern cartography deals with complex processes of geospatial information organization, access, display, and use. Collaboration and interactivity from both the cognitive and the usability perspective are well known research areas [Gah07].

Large-scale interactive Geographic Visual Analytics currently faces two major problems related to computer science:

1. **End-to-End Delay:** The traditional three-step workflow imposes a noticeable end-to-end delay between the query issuance and the retrieval of the final data ready for visualization:
   (a) A SQL query is issued to a spatial database.
   (b) The database returns Standard Open Format.
   (c) The Standard Open Format is converted into SVG.
2. **Read-Only:** The traditional workflow does not consider the aspect of collaboration as the user can not persistently enhance the existing data on-the-fly with individual attributes such as comments or even pictures.

Our contribution is fivefold:

1. **Two-Step Workflow:** We shortcut the traditional workflow by directly storing SVG data in a native XML database. This results in the following concise two steps eliminating the intermediate data conversion step as described with the problem statement:

   (a) An **XQuery expression** is issued to a native XML database.
   (b) The native XML database returns **SVG**.

2. **RESTful Interface:** We introduce a RESTful web interface to Geographic Visual Analytics to cleanly separate client and server in a standardized and scalable way.

3. **Temporal REST:** We extend Geographic Visual Analytics by an inherent temporal dimension which allows to query the current as well as all past versions of the stored SVG and its related data through the alluded easy-to-use RESTful interface.

4. **Read-Write:** We allow to interactively enhance the already stored SVG with statistical data and new attributes through XQuery Update.

5. **Implementation:** We provide a prototype implementation based on TreeTank to estimate the impact of eliminating the traditional intermediate data conversion step.

6.1.2 Background

Representations such as cartographic maps create links between representation and user interface and map user cognition and geospatial data. A variety of problem solving and data exploration tasks are addressed using cartographic representations. The ongoing technological development changes the representation forms, the spatial data handling, the related information science, the technology communities, and the potential of these representation forms for productive use. Effectiveness also is a theme discussed when using such representations and it is linked with the behavior of the user interacting with the display. The widespread availability of cartographic maps throughout the Internet leads to increased expectations on how to represent these maps [FAA+01].

Established specifications such as the eXtensible Markup Language (XML) as a technique for coding and structuring data should prove beneficial for portraying and interacting with geospatial information and visualization [FAA+01]. Many different approaches (server-side, client-side, hybrid) are available to improve the performance [CP06]. Each approach has its distinguished impact on data manipulation, map management, user interactivity, and the distribution of server-side or client-side tasks [CP06]. In addition, scalability and the option for collaboration also vary with each approach.

Scalable Vector Graphics (SVG), an open vector-oriented XML grammar, is suitable to visualize data. Dunfey [DGB06] proposed to use SVG to develop an open architecture for a vector GIS. SVG is a powerful tool and has the potential to visualize data now and in the future. SVG can be used to view vector graphics in a browser. There are plug-ins such as the Adobe SVG Viewer [Ado99] and an increasing number of browsers directly supporting SVG [DGB06]. Batik [Apa05], a Java SVG toolkit, allows to develop applications which use SVG for visualization. Neumann and Winter [NW09, DGB06] proposed to use SVG because it is an ideal vector format for web-based mapping. SVG, however, should be compliant with the OpenGIS Recommendation on the Definition of Coordinate Reference System for a XML grammar [W3C11]. SVG suffers from its lack to store additional attributes. Still, SVG is a desirable tool and the use of a separate XML file for storing additional information is preferred [DGB06].

With Geographic Visual Analytics, data is stored in a spatial database and can be exported as XML in a traditional geographical information system (GIS) file format or as SQL data. Widely used proprietary databases such as ESRI ArcSDE or Oracle Spatial store geospatial information in a binary long data type in an unpublished binary encoding. As such, the SVG document can only be extracted with the help of an SQL query. The traditional approach is to deliver the requested data in a Standard Open Format, e.g., a ESRI Generate File. An intermediate data conversion step is required to generate the SVG used for a flexible and easy-to-use interface. The traditional workflow is depicted in Figure 6.1 [DGB06].

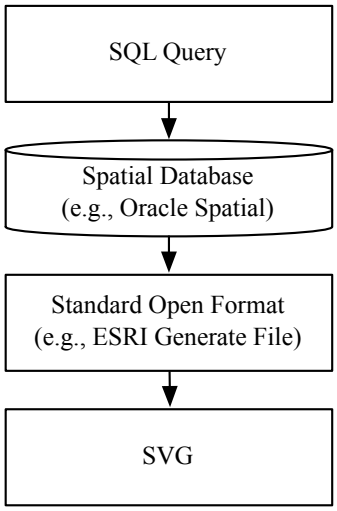


Figure 6.1: *Traditional three-step workflow converting the native output of a spatial database into SVG*

6.1.3 Streamlined Two-Step Workflow

We suggest a two-step workflow as depicted in Figure 6.2:

1. An **XQuery expression** is issued to a native XML database.
2. The native XML database returns **SVG**.

In stark contrast to the traditional three-step workflow, the intermediate data conversion step is eliminated, i.e., there is no need for converting the Standard Open Format such as an ESRI Generate File into SVG. The eliminated intermediate data conversion step both makes heavy use of CPU and I/O and mainly contributes to the large end-to-end delay virtually inhibiting interactive Geographic Visual Analytics.

At the hearth of our main contribution lies the switch to a native XML database capable of directly storing and emitting fine-grained XML data. Unlike traditional relational databases, native XML databases do not store the XML data as character large objects and inherently know about the XML structure and XML nodes. The finer granularity allows to answer complex queries and extract the stored XML in a scalable fashion because the parsing and reconstruction process required with character large objects can be omitted. In addition, most state-of-the-art native XML databases support modifications of the stored XML.

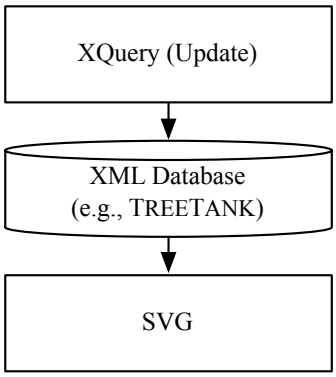


Figure 6.2: *Streamlined two-step workflow directly outputting SVG stored in a native XML database*

Notably, the native XML database must not necessarily store a single large SVG document but may separate the SVG, i.e., cartographic data, from other XML, e.g., statistical or user-provided data. The single XQuery expression issued to the native XML database is responsible to tell the system how to enrich the SVG with additional data, e.g., how to color different regions of the map due to the population density. The native XML database then executes the query by swiftly searching and combining all required XML fragments to finally return a single SVG. This lays the ground for distributing the underlying native XML database internally while still maintaining a single interface to the upper layers. Consequently, the elimination of the intermediate data conversion step by switching to a native XML database such as TreeTank leads to fundamentally better scalability than traditionally available.

6.1.4 RESTful Geographic Visual Analytics

Representational state transfer (REST) [Fie00] is a set of network architecture principles which outline how resources are defined and addressed. Practically speaking, REST defines a simple and scalable interface to exchange resources over HTTP. Each resource must be uniquely addressable through hypermedia links meeting a universal syntax. A well-defined and typically small set of HTTP operations specifies how to proceed with the obtained resource. The basic operations are POST to create a resource, GET to read a resource, PUT to update a resource, and DELETE to remove a resource.

The time-tested scalability and unquestioned expressiveness of REST makes it the interface of choice when it comes to handle large-scale SVG data. The clean separation of client and server at the web layer (HTTP) allows both sides to be independently implemented while drawing from state-of-the-art standardized web technologies such as Java, Ruby on Rails, or Adobe Flex. In addition, REST is a bidirectional interface both for querying and modifying the requested resource.

As the specification of the RESTful interface lies in the hands of each service provider, he can both specify the set of available operations and resources. Additionally, the implementation be it the traditional three-step or the streamlined two-step workflow can be exchanged without modifying the RESTful interface and breaking existing clients.

A RESTful HTTP request looks like:

```
GET http://{host}/{path-to-resource}?{query}          (6.1)
```

where {host} is the name of the web server hosting the Geographic Visual Analytics service, {path-to-resource} is the name of the resource to retrieve, and {query} is a query expression used for a detailed specification of the result to return. E.g., a simple request might be:

```
http://localhost/MapOfAmerica?PopulationDensity          (6.2)
```

to retrieve the SVG representing the map of the United States of America colored with the population density. Note that the query must not necessarily be an XQuery expression but it may be an implementation-independent expression which is mapped to XQuery (or to a SQL query) on the server side.

6.1.5 Temporal Geographic Visual Analytics

Figure 6.3 depicts the evolution of a cartographic map over time. Currently, the user usually retrieves the version of the map as it was last stored. A simple temporal extension to the RESTful interface (see Chapter 5) empowers the user to retrieve the map as it looked like at any past point in time. This enormously facilitates interactive Geographic Visual Analytics in a temporal fashion. E.g., the evolution of the population density of a whole region can now easily be visualized.

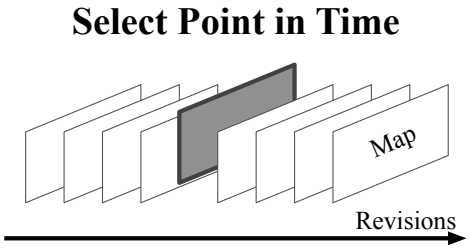


Figure 6.3: *Temporal selection of a cartographic map as it looked liked at a past point in time*

A temporal RESTful HTTP request looks like:

```
.../{path-to-resource}/({point-in-time})?{query}          (6.3)
```

where ({point-in-time}) denotes a point in time either specified as a version number or an ISO-compliant date. The parenthesis '(' and ')' mark the temporal expression which can easily be extended in the future, e.g., to support time periods in addition to single points in time.

While the temporal dimension comes along with an unimposing but convenient extension to the well-known RESTful interface, it potentially imposes a huge storage and processing overhead if the underlying system does not natively support temporal queries. Our native XML database TreeTank was designed to support temporal queries out-of-the-box and is able to retrieve any past version at the same cost as the last version.

6.1.6 Case Study

Google Maps is an impressive example how users can not only visualize geographic data but also enrich it with new information. Unfortunately, the traditional three-step workflow does not support a feedback channel enabling user-driven enhancements or additional attributes. The complexity lies in the question where to efficiently store the newly added attributes, how to query them, and how to merge them with the existing geographic information.

Our streamlined two-step workflow leaves a single point to store the user enhancements for later retrieval: the native XML database. The user issues an XQuery Update expression which tells the native XML database to insert, change, or delete an attribute. The modification immediately becomes visible for reading queries and therefore optimally supports collaborative projects. Taking the RESTful interface into account, the user may also issue a generic query by calling a POST, PUT, or DELETE operation which is transformed into a XQuery Update expression on the server.

To pick up our example about the population densities of a given region, we can now give a complete example involving all mentioned contributions. One can imagine a RESTful service running at the local host. Initially, the service does not contain any data at all. Then, the user successively performs the following actions through the RESTful web interface of the service:

- The SVG-based map is uploaded to the server.
- Attributes are inserted into the SVG to denote several regions by their identifiers.
- The population density data for the geographic region of the SVG-map is uploaded. Each upload contains the data of a specific year.
- More SVG layers, e.g., rivers or streets, are uploaded to the server.
- The user tags several regions on the map with his own data by storing more attributes for each region.

6.1.7 Conclusions and Outlook

Preliminary measurements on a state-of-the-art desktop computer show two significant advantages of TreeTank. First, it compresses the original XML data while storing it in its native data structure. Second, it allows to quickly retrieve the original XML. Table 6.1 shows the promising preliminary results of both the compression and time measurements for three maps of different sizes. The excellent compression ratio is due to the verbosity of SVG. The time of the data conversion step alone (excluding the time to retrieve the original data from the spatial database) takes much longer than the time required to retrieve the whole SVG from TreeTank.

Both cartography and computer science mutually foster their respective research. Interdisciplinary fields such as Geographic Visual Analytics extensively draw from the melting pot of both a comparably experienced and a youthful science. While one needs ever faster tools to interactively visualize large data sets, the other is in need for a catchy use-case.

	SVG Size [MB]	TreeTank **Size** [MB]	**Conversion Step Time** [s]	TreeTank **Time** [s]
Map 1	0.02	0.01	0.38	0.22
Map 2	2.60	0.21	5.52	0.79
Map 3	138.70	11.30	102.38	3.79

Table 6.1: *Preliminary measurements with* TreeTank. *The conversion step time is the time required by the third intermediate step eliminated with our streamlined two-step workflow. It can be seen, especially for larger maps, that the workflow is speed-up from formerly minutes to seconds for a single interaction*

We identified the data conversion step of the traditional workflow of Geographic Visual Analytics as a major bottleneck imposing large end-to-end delays between the query issuance and the retrieval of the final SVG ready for visualization. We suggest a new streamlined two-step workflow based on a native XML database promising better scalability and diminished delays while better supporting the interactivity aspect from the end-user perspective. We propose to use a RESTful interface providing scalable temporal read-write access to geographic as well as statistical or individual data. The clean separation of client and server at the HTTP web layer assures back-wards compatibility and better extensibility. We give an example on how to effectively use the RESTful interface and provide preliminary measurements based on our prototype implementation TreeTank. The excellent compression ratio as well as the fast SVG delivery strongly encourage further research.

Future work will include a detailed elaboration and implementation of our ideas to prepare a solid foundation for the research community of Geographic Visual Analytics. While the RESTful interface might look the same for quite some time, it will independently foster the progress on both the client and the server side. Under the hood, we will see distributed native XML databases capable of storing and querying ever larger sets of geographic and statistical data while supporting a multitude of concurrent collaborating users. Five steps towards large-scale interactive Geographic Visual Analytics – a leap forward from the computer science perspective.

6.2 Collaborative Geographic Visual Analytics

6.2.1 Introduction

Geographic visual analytics (GVA) is a highly interdisciplinary research field, with tight links to different related disciplines, and having needs and interests in synthesizing information and deriving insights from massive, dynamic, ambiguous, and often heterogeneous data sources [KMSZ06]. The scientific objective of GVA is to understand how both individuals and teams carry out analytical reasoning and decision-making tasks based on complex information, and to use this understanding to develop and assess information and communication technologies for this purpose [MCM+06].

Increasing the sizes and complexities of data sets being collected, handled, and analyzed by visual analytics experts calls for new cross-disciplinary approaches [AAD+08]. Efficient and effective storage and exchange of very large and complex distributed spatio-temporal databases is not only an important enabler for GVA,

but also a research focus of the database research community within computer science. While previously large geographic data sets were typically of structured alpha-numerical nature (i.e., remote sensing images, census data sets, etc.) more recently GVA researchers have had to work with a multivariate mix of structured (relational) databases and increasingly semi-structured (e.g., XML-based) and unstructured (e.g., plain text) data sets, all readily available on the Internet.

A flexible and dynamic data storage and access infrastructure is especially needed when representing movement, dynamism, and change [AAD+08]. Ideally, GV analysts should have efficient tools at hand for interactively access, rapidly modify, exchange in real-time, or generate entirely new representations on the fly from underlying massive data sets, whenever the research context requires it.

Today, GVA user interfaces establish the necessary linkages between collected geographic data sets, data representations stored in databases, as well as with external (graphic) visualizations presented to a user which interact with internal (mental) representations. Ongoing technological developments provide continuously changing data types (i.e., from tracking devices, LBS, sensor networks, audio, etc.), which in turn require new data-handling structures for efficient GVA.

According to MacEachren and Kraak [MK01], one of the challenges is to develop extensible methods and tools that enable the understanding of, and insights from, increasingly large and complex volumes of geospatial data that are becoming readily available. Scalability of GVA solutions has become one of the bottlenecks when dealing with massive databases.

Many different (server-side, client-side, hybrid) data-handling approaches are already available for Internet-based geographic information systems (GIS), and their goal is to improve data access performance [CP06]. Each approach has its specific advantages and disadvantages with respect to data manipulation and management, user interactivity, and the distribution of server-side or client-side tasks [CP06, YZ08]. Scalability and the provision of distributed collaboration varies significantly with each approach.

One of the main challenges for highly interactive and distributed GVA is the inherent potential for media breaks when dealing with distributed and diverse databases, thus reducing the potential for knowledge discovery. For example, knowledge might be disseminated through one media channel (such as written communication) in the form of emails or a journal article summarizing insights from a database that is no longer directly accessible from this particular media channel. Essentially, the media break is enforced by the underlying data infrastructure, as this infrastructure does not natively support the dynamic adaptation of large-scale data sets to various media channels. Each media break within a collaborative research context hinders knowledge discovery as it requires the (manual) conversion of data from one format to the next. The preparation, conversion, and reviewing steps all require time and significant computational resources when dealing with massive data sets. Consequently, real-time or interactive collaborations over a network are severely hindered.

This section presents a data storage and a visual access framework capable of dealing with large-scale and frequently changing semi-structured (XML-based) spatio-temporal data sets being increasingly used in GVA research contexts. The proposed GVA infrastructure enables analysts to access and modify large and complex data sets and rapidly display these changes in response to user actions, thus enabling efficient and collaborative visual data exploration environments [AAD+08].

Specifically, we propose an XML-based infrastructure to reduce the potential number of media breaks within geographic visual analytics. Our infrastructure provides sound support to securely store and quickly access dynamically changing data, thereby providing adequate cognitive knowledge in a scalable, web-based, and collaboration-oriented way. We describe the underlying technology and provide a case study to demonstrate its benefits.

6.2.2 Background

Recent developments foster the integration of data storage and display technologies in ways not possible before. The (well designed) web-based geo-visualization display has become an interface to massive, complex and distributed databases that can support efficient information access and knowledge construction.

The Open GIS Consortium has initiated web mapping interoperability initiatives and specifications to develop interface specifications for geographic data [OGC97]. This includes the Geography Markup Language (GML) encoding standard to express geographic features [GML98], or the Web Feature Service (WFS) Implementation Specification for retrieving geographic features across the web [OGC02]. In addition, geographic features stored in this fashion can be displayed using the Scalable Vector Graphics (SVG) format, an open standard developed by the world wide web Consortium (W3C) [PZ04]. SVG makes use of the eXtended Markup Language (XML) to describe two-dimensional geometric objects (points, lines, and polygons). In Neumann and Winter's [NW01] words, XML is the future core-technology for all upcoming web standards.

Peng and Zhang [PZ04] have outlined the role of GML, SVG, and WFS in building an Internet-based geographic information system (GIS). Open issues were in their opinion the compression of GML and SVG files, seen also as the easiest issues to solve. More complex open issues are the client-side SVG user interface and data processing tools to assist users as they interact with GML data. More recently, Yao and Zou [YZ08] highlighted interoperability challenges of Internet mapping tools based on the open source approach. A core challenge is the efficient transfer of data between relational and object-oriented databases. For example, widely used proprietary databases such as ESRI ArcSDE or Oracle Spatial store geospatial information in a long, binary data type in an unpublished format. To access these data for display with SVG, an SQL query is required. The traditional approach is to deliver the requested data in a Standard Open Format, e.g., an ESRI Generate File. An intermediate data conversion step is then required to generate the SVG document from the ESRI Generate File, before it can be presented to the user in the form of an easy-to-use graphical interface [DGB06].

According to Neumann and Winter [NW01], databases are easier to query or update while XML is perfect for data exchange and archiving. Note that this statement was made back in 2001 due to the lack of linearly scalable native XML databases with logarithmic update characteristics. It shows the urgent need for concepts such as TreeTank.

SVG displays can be constructed directly out of (XML) database and be presented to a user for interactive geo-visualization and visual analytical knowledge construction. SVG is optimized for graphic rendering on the web. Features such as vector display, animation, interactivity, transparency, graphic filter effects, shadows, lighting effects, and easy editing are all provided with SVG [YZ08]. However,

while SVG is eminently suitable for graphic content delivery by providing flexibility for user interactions [NW01], one should recognize the problem of missing topology for advanced spatial analysis and such limitations in cartographic symbolization as missing complex line styles.

6.2.3 Approach

We propose a web-based, flexible, and scalable GVA framework using native, XML-based data storage and back-end handling infrastructure coupled with SVG at the system-user interface. This GVA infrastructure provides analysts with highly interactive GVA tools to support complex data exploration and decision-making tasks. It includes flexible data depiction, high computer-user interaction, and collaboration over the web.

We favor SVG for our approach, as it allows for rapid system development and prototyping, provides fast response times for interactive query requests, and supports efficient data interoperability over networks [YZ08]. Similarly to Yao and Zou [YZ08] and Dunfey et al. [DGB06], we expect that SVG will be supported natively in most if not all web browsers, and thus no extra plug-ins will be necessary.

We natively store SVG data in an XML-based database, even though other authors have argued against using SVG as basis for geovisualization [YZ08], because it is not suitable for securely and efficiently storing, managing, or delivering spatial data over the network. We argue that TreeTank solves such problems as secure and efficient storage, management, and network-based data delivery. Another XML-based language, the Geographic Markup Language, specifically targets geographic data. Fortunately, SVG and GML are highly compatible and can work in synergy. For example, Yao and Zou [YZ08] convert GML-based data to SVG before transmitting data to the client for display.

We employ the representational state transfer (REST) technology for queries to, and feature extraction from, our XML database. REST is a set of network architecture principles which outline how resources are defined and addressed. Practically speaking, REST defines a simple and scalable interface for exchanging resources over the Internet using the HTTP protocol. Each resource must be uniquely addressable through hypermedia links, meeting a universal syntax. A well defined and typically a small set of HTTP operations specifies how to proceed with the obtained resource. The basic operations are POST to create a resource, GET to read a resource, PUT to update a resource, and DELETE to remove a resource. The scalability and unquestioned expressiveness of REST makes it the interface of choice when it comes to handling large-scale SVG data on a network. The clean separation of client and server at the web layer (HTTP) allows both sides to be independently implemented, while drawing from state-of-the-art standardized web technologies such as, Java, Ruby on Rails, or Adobe Flex. In addition, REST is a bidirectional interface both for querying and modifying the requested resource [Fie00].

6.2.4 Infrastructure

At the heart of our contribution lies the switch to a native XML database capable to directly store and emit fine-grained XML data. Unlike traditional relational databases, native XML databases do not store the XML data as character large ob-

jects (CLOB) and inherently know about the XML structure and XML nodes. The finer granularity allows answering complex queries and extracting the stored XML in a scalable fashion because there is no parsing and reconstruction as required with character large objects. In addition, most state-of-the-art native XML databases support modifications of the stored XML.

Our XML-based infrastructure consists of two components, i.e., the TreeTank storage manager as described in Chapter 3 and Chapter 4 as well as the web interface Temporal REST described in Chapter 5. The two components are connected to implement a two-step workflow as follows:

- An **XQuery expression** is issued to TreeTank through Temporal REST.
- TreeTank returns **SVG** through Temporal REST.

In stark contrast to the traditional three-step workflow based on relational spatial databases, the intermediate data conversion step is eliminated, i.e., there is no need for converting such a standard open format as an ESRI Generate File into SVG. The eliminated intermediate data conversion step makes heavy use of CPU and I/O, which contributes to large end-to-end delay, thus virtually inhibiting interactive Geographic Visual Analytics.

Temporal REST

While there exists a variety of solutions to access XML resources over the web, there is – to our knowledge – no generic and unified solution to conveniently access:

- The **current version** of the XML resource or any subset thereof.
- The **full version history** of the XML resource or any subset thereof.
- The **full modification history** of the XML resource or any subset thereof.

We decided to work with XML as a fine-grained tree of nodes and evolve this tree over time through user modifications. As such, we realize that we can access single nodes or whole sub-trees, i.e., XML fragments, within a temporal dimension in a unified, scalable, and robust way.

Only if we consider the entire life cycle of an XML resource, including the past versions and the (transaction-based) modification history, will we get a complete idea of its true power. Notably, collaboration processes frequently involve asynchronous workflows. As such, the effectiveness of the workflow largely depends on the ability to highlight the modifications which took place during the last (or any past) step of the workflow.

We use Temporal REST with its related protocol message exchanges to generically implement our idea of exploiting web-based XML resources. Based on the Pareto principle, our proposal is simple enough for the average web application developer and at the same time it is extensible enough to be used with complex setups.

There are three different ways of accessing nodes and subtrees (XML fragments) in an XML resource. These include, first, the step-by-step tree navigation (XPath), second, the query including joins and other complex expressions (XQuery), and,

third, the ID-based random node access (DOM). Temporal REST supports all three and complements them with a temporal expression as described later. Note that XPath is a subset of XQuery.

The select operation allows the retrieval of a sequence of items as defined by XQuery. Each item either is an atomic value, or an XML node, or a modification event. The selection can be query-based or REST ID-based. Temporal REST will restrict the execution domain of both the query and the REST ID according to the temporal expression by either selecting a point in time or a time period. While a query may return a sequence of multiple items, an access solely based on a REST ID will return a sequence with at most one item. If the query and REST ID approach are combined, the query treats the node with the given REST ID as the root node of the query. The query-based approach makes it possible to add new query languages in the future and express complex queries, including operations such as full-text search or joins. The REST ID-based approach makes it possible to directly select an item with optimal performance because the system does not have to compile and optimize the query.

The temporal expression must be enclosed with round brackets '(' and ')' and contain a single point in time or a time period consisting of two points in time separated by a dash '-'. A point in time can be a version number, an ISO date in short notation, i.e., without dashes or colons, or nothing. If no date or version is provided, the last successfully committed version is selected. Note that the ISO date in short notation is compliant with the specification of a URL. A single point in time will retrieve the XML fragments as they were at the given version. The time period will retrieve the modifications between (and including) the two provided points in time in ascending or descending order. Leaving out the temporal expression automatically causes a fallback to the last successfully committed version for backward compatibility. Table 6.2 shows the HTTP request and response required to either select a single point in time (Example 1) or a time period (Example 2).

	HTTP Request	**HTTP Response**

1	`GET http://../document/(1)?`
	`  //para/text()`
		`<?xml version="1.0"?>`
		`<rest:response xmlns:rest="REST">`
		`<rest:sequence rest:revision="1">`
		`<rest:item>`
		`Joe is happy.`
		`</rest:item>`
		`</rest:sequence>`
		`</rest:response>`

2	`GET http://../document/(2-3)`	`<?xml version="1.0"?>`
		`<rest:response xmlns:rest="REST">`
		`<rest:sequence>`
		`<rest:item rest:revision="2">`
		`<para` **`rest:id="3"`**`>`
		`Mike is happy.`
		`</para>`
		`</rest:item>`
		`<rest:item rest:revision="3"`
		`rest:id="2"/>`
		`</rest:item>`
		`</rest:sequence>`
		`</rest:response>`

Table 6.2: *Two example REST request and response pairs. Example 1 retrieves an XML sub-tree at version one. Example 2 retrieves the modifications on the whole document during versions two and three*

A single node or a whole sub-tree can be inserted either as the first child of an existing node or as its right sibling. As such, the insert operation requires a query selecting a number of nodes or a REST ID besides the actual XML fragment to complete the insertion. During the insertion process, the back-end system will assign REST IDs as described above. Note that the insertion of an attribute must be made with the PUT operation which changes the whole node.

A single node can be replaced with or without the replacement of its sub-tree. Again, the updating operation requires a query to select a number of nodes to update or a REST ID. In addition, the actual updated XML fragment has to be provided. Restricting the effect of the update to the node (not effecting its sub-tree), allows the insertion of an attribute into an existing node without changing its whole sub-tree.

Whenever a node is deleted, the node and its sub-tree are purged from the system (but not from the past versions). The deletion operation requires a query or a REST ID to select the nodes to delete.

TreeTank

We tried to implement Temporal REST on top of existing open and closed source technologies. Unfortunately, it turned out that there was no file system, no relational database, and no native XML database to efficiently support all our requirements at once. All systems struggled with the combination of large-scale, heterogeneous, and version-based data. In fact, it was possible to mimic the versioning feature, i.e., to keep past versions for given data. However, the resource consumption with respect to disk, memory, and CPU already exceeded the capabilities of state-of-the-art systems even for data sets far smaller than a single gigabyte. Eventually, we decided to use TreeTank. Three systems mainly influenced our work on TreeTank. The ZFS [BM04] file system handles transactions and snapshots but still operates at file-level granularity, which is far too coarse for small-grained XML data. The version control system Mercurial brought along Revlog [Mac06], a space-efficient method to store all past versions – again only at file-level granularity. XPathAccelerator [Gru02] inspired the low-level XML encoding of TreeTank as it makes it possible to work efficiently with read-only large-scale heterogeneous data sets.

TreeTank is a native XML database designed to provide scalable read and write access to XML data. TreeTank concurrently allows multiple read transactions and a single write transaction each of which creates a new version per transaction commit. TreeTank was designed to be secure and easy to maintain. The scalability of TreeTank results from the concurrent use of resources such as processing and storage units and from the design of the main internal data structure to store the XML tree.

The decision to only support a single write transaction at any time makes it possible to run any number of processes concurrently, accessing any past versions or modifications. The newly modified data are clearly separated and only become visible after the last successful transactional commit to processes different from the write transaction process. If multiple users want to work on the same XML tree at the same time, a transaction manager is required to coordinate, i.e., sequentialize the changes, or a workflow has to be established stating clearly when each user is allowed to work and what he or she can do. Alternatively, a locking scheme has to be established which may follow an optimistic or pessimistic locking policy. However,

it turns out, that in many real-world use cases, only a single user is working on a given part of the tree at any time, or that the natural workflow of a team working with XML data resolves modification conflicts before they even could appear.

Each XML resource, i.e., an SVG file, is bound to a session. The session is allowed to start transactions. Read-only transactions support two different selection modes. The first selection mode makes it possible to answer the question of what the data looked like at a given point in time. The second selection mode leads to an answer to the question of what changed between two different points in time. A write transaction can select the last successfully committed point in time and modify it. Note that TreeTank currently does not support checkpoints within a single write transaction, i.e., the modifications on an XML tree are made persistent at once during the commit operation. A rollback can revert the XML tree to any past version.

The data structure of TreeTank was optimized for updates. At most three directly related nodes must be updated whenever a single node or sub-tree is modified. Only the modified nodes are stored on disk in a compressed page. Note that traditional databases usually store the whole page (which may potentially contain dozens of nodes) even though only a single node may have changed. Still, care has to be taken that reads do not have to collect a huge number of scattered changes to reconstruct a single page. We opt to intermittently store a snapshot of the whole page to also support reads with reasonable performance. Compressing all pages, storing only the page modifications, and intermittently storing snapshots of the pages all help to reduce the storage requirements by one order of magnitude. As a result, TreeTank does not consume significantly more space and it can swiftly reconstruct any past state or modification.

Security is not a choice with TreeTank. Care was taken to implement only time-proven cryptographic primitives with sufficient key lengths and well chosen cryptographic modes so as not to create a weak link which could be attacked to break the whole system. TreeTank encrypts all compressed pages before they are stored on disk. This guarantees the confidentiality of the stored XML tree, even if the TreeTank files are exposed to the public or transferred through insecure networks. Besides the encryption, a strong message authentication code is derived from each compressed page and stored with a reference to the page. As each reference contains the message authentication code of all its children, the integrity and authenticity of the whole TreeTank can be verified recursively. The root message authentication code can be securely signed and further secured by an external secure time stamping mechanism, which also ensures that modifications cannot be denied. The availability of TreeTank can be guaranteed on the application level by a master-slave replication which consumes very little network bandwidth and is perfectly suited for geographically distributed operations. The master-slave setup ensures that all modifications applied to the master are synchronously or asynchronously propagated to the slave. The tight integration of security enables storage of sensitive data in the TreeTank. This is especially important because visualizations are usually based on large data sets collected from the internal operation of an organization or project and must not be exposed to the public.

Preliminary measurements on a state-of-the-art desktop computer show two significant advantages of TreeTank. First, it compresses the original XML data while storing it in its native data structure. Second, it enables a fast retrieval of the original XML. The promising preliminary results of the compression and time measurements for three SVG files of different sizes are as follows:

- The size of the TreeTank is up to ten times smaller than the original SVG file and TreeTank can deliver the original SVG data up to twenty times faster than a relational data-base with spatial extensions.

- The excellent compression ratio is due to the verbosity of SVG.

- The time of the data conversion step alone (excluding the time to retrieve the original data from the spatial database) takes much longer than the time required to retrieve the whole SVG from TreeTank.

6.2.5 Case Study

In this section, we provide a case study to demonstrate not only the feasibility but also the significant benefit a user can gain from our infrastructure. Most importantly, we want to build a mindset for designing and using our infrastructure because it is notably different from traditional workflows both on the technical and application levels. With our infrastructure, the user can organize and later modify the data in the XML tree, as he or she likes. He can mix document-centric sub-trees containing information, e.g., in the OpenDocument format, with sub-trees compliant with ready-to-visualize SVG data, as well as data-centric statistical information. Figure 6.4 shows a typical setup of our XML-based infrastructure.

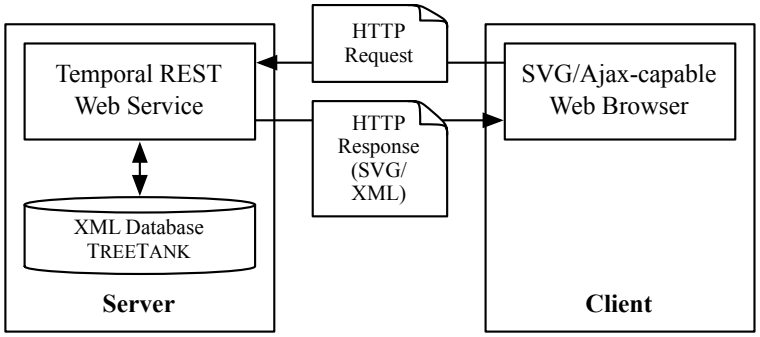


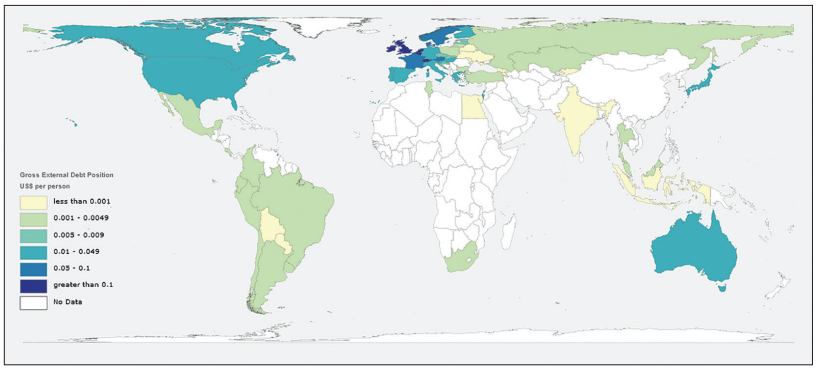
Figure 6.4: *A typical setup of our XML-based infrastructure. Server and client exchange SVG/XML data with HTTP-based Ajax technology*

In this case study, we build an example TreeTank of gross external debt positions in U.S.$ per person. This information is available on a quarterly basis [Wor08] and perfectly suited to illustrating how a team can create sophisticated visualizations based on a set of statistical data. Four versions of the visualization can be seen in Figure 6.5.

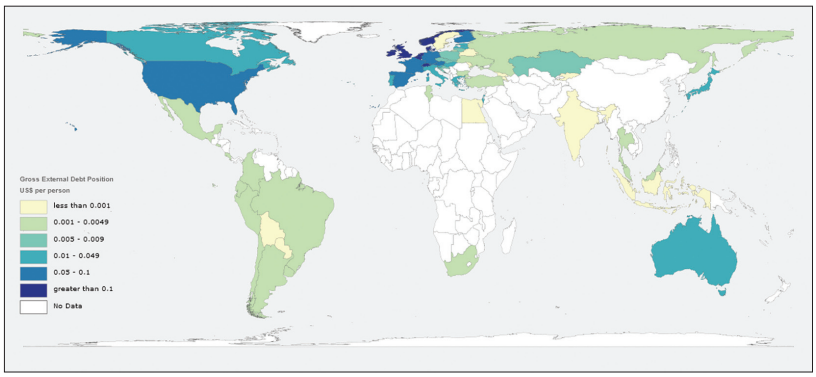
Note that the TreeTank is exposed to authorized users through a web service running Temporal REST. While we intentionally present a basic example, our infrastructure can deal with any large-scale heterogeneous data as long as the data can be transformed into XML.

The first step is to convert the Excel-based statistical information into a data-centric XML. This is a straightforward step and only required if the original data are not available as XML. The resulting XML can be directly imported into TreeTank by inserting the whole XML document through Temporal REST. We can now query Temporal REST to extract the whole document or any sub-tree therein.

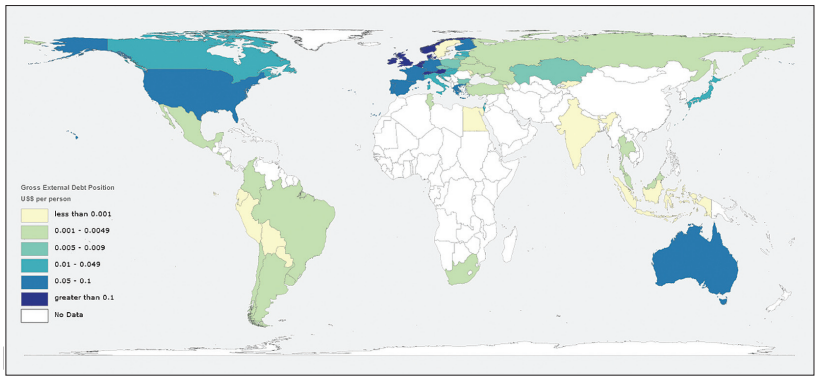
(a) The SVG sub-tree with the map of the world and a description box



(b) The gross external debt positions in U.S.$ per person for the year 2006; third quarter



(c) Same information as in b) but for the year 2007; third quarter



(d) Same information as in b) but for the year 2008; third quarter

Figure 6.5: TreeTank *of gross external debt*

For the second step, we need an SVG representation of the world with all countries. One can rely on open source SVG world maps or retrieve an individually configured world map from a traditional relational spatial database, depending on requirements. To keep the statistical data separate from the SVG data, we insert the new node *statistic* as the parent of the statistical XML data as shown in Figure 6.6. Then, we insert a new node *geodata* as the right sibling of *statistic* and group the two nodes *statistic* and *geodata* under the third new node *example*. We then insert the whole SVG data under the node *geodata*. Hence, we can retrieve the plain statistical data by selecting the sub-tree rooted at *statistic* or visualize the world map within any SVG-enabled web browser by selecting the sub-tree rooted at *geodata*. To combine the statistical data with the visualization, we have to make sure that both sub-trees store the ISO country codes for each country. If this is not already the case, we can update each country in each sub-tree. Note that most SVG-based world maps will separately store an SVG path for each country.

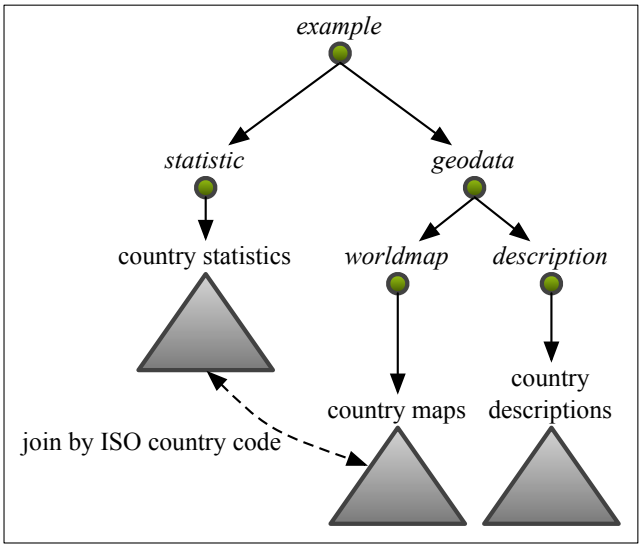


Figure 6.6: TreeTank *storing worldmap XML tree alongside with statistical data, additional descriptions, and SVG layers*

Meanwhile, we created a set of versions, each consisting of a Temporal REST modification request. At any time, we can retrieve an older version or list the modifications applied to past versions. This is convenient, if one wants to know what changed, e.g., in the sub-tree under *geodata*. It is also assuring to know, because one can revert the tree to a past version if an unintended modification took place. At no time, data are overwritten or lost. Furthermore, the author of the changes can provide commit comments with each Temporal REST modification request to document his intentions and the evolution of the tree.

We prepare the visualization of statistical information by defining value ranges and color schemes for each value range. Then, we add the color information as an XML attribute to each element in the *statistic* sub-tree based on the statistical value and make sure that the statistical information is grouped in sub-trees for each year and within the years for each quarter. Next, we add the SVG elements to the *geodata* sub-tree required to draw a box displaying the color scheme and value ranges. To better support layering in SVG, we group the SVG elements required to draw the box under the XML element *description* and then group the SVG path elements required to draw the world map under the XML element *worldmap*. This step helps

to interactively enable or disable layers and can later be extended to support, e.g., layers containing water bodies, charts, or other GUI elements required for improved and convenient user interaction. To prepare the coloring of the countries according to the selected statistical data, we add the appropriate SVG color attribute to each path element. Finally, we add a SVG GUI element under *geodata*, which enables us to interactively select a quarter of the year.

The actual procedure to color the world map according to the selection can either be implemented with an XQuery expression issued through Temporal REST or with JavaScript on the client side. If XQuery is chosen, one must select both the *statistic* and the *geodata* sub-trees and then set the color attribute of the SVG path elements to the color attribute of the statistical data by joining them by country code. When JavaScript is preferred, both the *geodata* and the sub-tree containing the statistical information for the selected quarter have to be transferred to the client and then joined together by looping through all countries and setting their color to the color value found in the statistical data. Note that the statistical data can be reloaded efficiently and on demand with Ajax technology.

The main differences between the XQuery and the JavaScript variant is the location where work is done (i.e., on the client or the server side) and the amount of data that has to be transferred over the network. In the case of XQuery, the join is calculated on the server side for each request. Then the result is transferred to the client and immediately visualized. In the case of JavaScript, large amounts of data have to be transferred to the client for the first request in order to calculate and visualize the join. For later requests, only the new statistical data are transferred, joined, and visualized. Thus, JavaScript is the better choice if the workload consists of multiple selections for different quarters. However, note that current JavaScript runtime environments are so slow that the XQuery variant might be faster even though all data for the visualization have to be transferred for each request. This may change in the near future since most JavaScript runtime environments are currently undergoing major rewrites to speed them up significantly.

An alternative to the method of joining pre-calculated persistent coloring information with the map is the purely dynamic calculation depending on the current user requirements. Again, the calculation can be performed on the server or on the client side, with the same advantages and limitations as noted for the join method.

Figure 6.7 gives an additional example of laying out GUI elements with SVG (including a sample chart) and was implemented by [Gia09]. The GUI elements of interest are the version slider, the map layer control, and the search field. All events are handled by JavaScript which uses Ajax technology to fetch missing data from the Temporal REST web service. The JavaScript itself is embedded in CDATA sections of the XML. In our view, this is not the most elegant way to store JavaScript in CDATA sections. However it is a straightforward and practical solution, which automatically guarantees the version of the application code itself. There already are technologies such as the XML user interface language (XUL) or Adobe Flex which describe GUIs and their interactions based on pure XML. More work is however needed before these solutions can be included in off-the-shelf web browsers.

We have shown that the XML tree can be grown exactly according to the user's demand. All relevant data sources can gradually be integrated with TreeTank and then queried and further modified from within one single infrastructure. While the last paragraphs only considered a single user performing the modifications, we describe the collaboration of multiple users collectively working on the same TreeTank in the next paragraphs. Note that each user can modify the XML tree

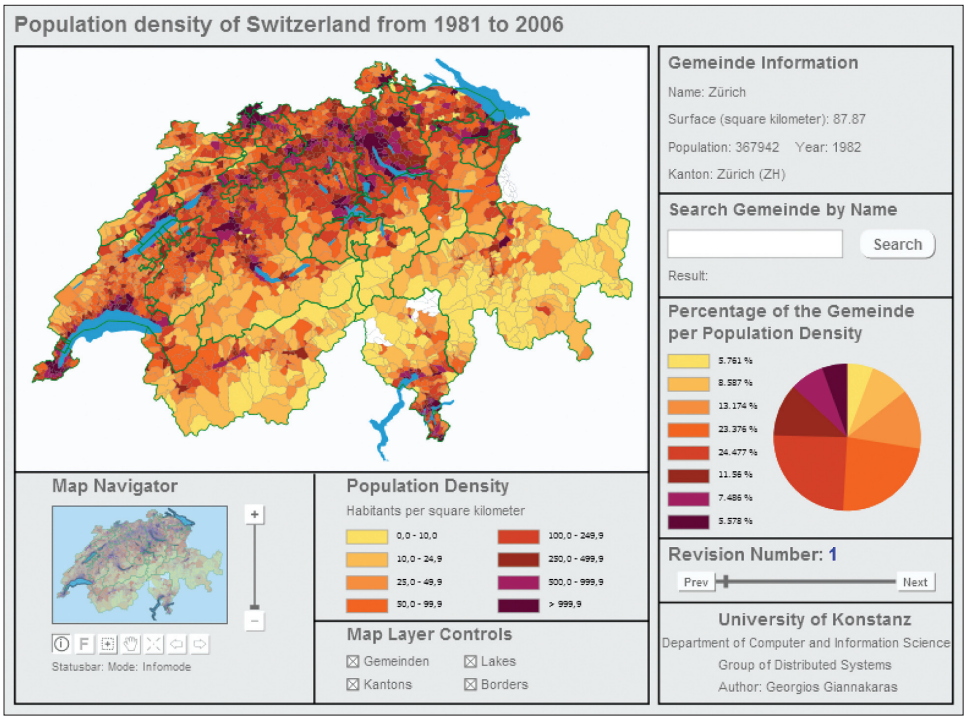


Figure 6.7: *An example of a rich SVG GUI providing a chart and extended end-user input facilities. Note that this additional SVG sub-tree could be plugged-in seamlessly with the existing* TreeTank

and add more statistical data or visualization elements as described before. As any professional publication or authoring workflow, it is important, however, that each user behaves according to a policy. With TreeTank and Temporal REST, this means that concurrent modifications have to be done in disjoint sub-trees.

While the current version of TreeTank does not provide a facility to enforce this behavior, it can be implemented technically on the application layer or non-technically in the organizational structure. We suggest a hierarchical responsibility delegation scheme, such that at any time, one author (person or process) is responsible for a given sub-tree unless he or she delegates a descendant to someone else with the option to revoke.

This scheme can be extended by a nonhierarchical access-control-list-based (ACL-based) scheme if required. To clarify the hierarchical responsibility delegation scheme, we imagine a situation where author A1 adds more statistical data each quarter, author A2 works on the SVG-based GUI and color schemes [MT94], and author A3 links the visualizations to scatter plots or other statistical graphics [AA99]. One possible hierarchical delegation then is as follows. The owner of the TreeTank delegates *statistics* to A1 and *geodata* to A2. A2 creates a new node *charts* and delegates it to A3. Then, all three authors concurrently modify the tree and will never cause isolation conflicts such as lost writes or dirty reads as they are stated in the ACID model, which is one of the oldest and most important concepts of database theory. Lost writes are prohibited by design because each author works in his responsibility domain, i.e., a dedicated sub-tree. Dirty reads are avoided because each author will only see successfully committed changes and has the option

to query the tree as it was at a given, fixed version. Whenever the user wants to switch to a newer version, he or she first checks for modifications on the sub-trees of interest and whether they impact his own work, e.g., introduce an inconsistency because the color attributes were dropped. Finally the user can adapt his or her part of the tree to the modifications.

6.2.6 Conclusions and Outlook

The findings from the case study open a wealth of opportunities for the end-user as well as an array of research challenges. The immediate benefit of our XML-based infrastructure is the very efficient use of processing and storage resources. Much more user requests can be handled per time unit, and the degree of interactivity is dramatically improved as the user actions are no longer a matter of minutes but seconds. Both throughput and interactivity are essential for collaboration-oriented environments where end-users are used to interact in an asynchronous as well as a synchronous fashion. The support for the evolutionary growth of tree (XML) data structures as well as the ability to store and query statistical and SVG data, side-by-side, help to reduce unnecessary media breaks which hinder the dissemination of (visually) discovered knowledge.

The research challenges are manifold. One challenge is to find and categorize tree structure and tree design patterns. Our infrastructure makes it possible to store huge amounts of unstructured data in a single TreeTank. Without patterns, the TreeTank could end up being a junk room where everything is contained but rarely something can be found in time. Hand in hand with the patterns comes the question of how best to organize and manage the concurrent access of multiple users assuming changing roles. In our case study, we suggested an organization form natural for tree-based data structures. But there may be other more efficient solutions. As with the tree structure and tree design patterns, the collaboration-oriented (authoring) workflows have to be collected, categorized, implemented, and tested with real teams. From a technical point of view, the challenge arises to integrate various indices with TreeTank to speed up specialized queries such as full text queries or spatial queries on rasterized data. While the server side can be further speeded up with the help of indices, the client side GUI and JavaScript environments still need to be revised to unleash the processing power of modern desktop or notebook computers. The GUI functionality of browsers and SVG plug-ins is not yet on par with native applications. Even the extensive use of Ajax and JavaScript does not hide the current shortcomings.

The case study made the assumption that there are multiple users but only one single TreeTank. When multiple teams concurrently grow their data structures in independent TreeTanks, the issue is how all these distributed TreeTanks can be integrated into one unified storage. While our infrastructure solves this by integrating different data sets into one tree, it does not yet provide support for integrating multiple trees into a forest.

We propose a new streamlined two-step GVA workflow for efficient data storage and access based on our native web-based XML database TreeTank and couple it with an SVG graphical user interface for visualization. Not only does our XML-based infrastructure substantially reduce access delays due to the elimination of intermediary data format conversion steps. Rather, it extends the user's options by providing significantly better scalability, inherent data security, and, most importantly, the ability to collaboratively work in GVA environments thanks to

optimized update support. With up to twenty times shorter data access delays and up to one tenth of the traditional storage requirements, our infrastructure improves interactivity and flexibility from an end-user perspective.

Furthermore, our infrastructure suggests a paradigm shift leaving behind dispersed disconnected data sets and media breaks and introduces a tightly integrated unified storage for complex spatio-temporal data sets of structured, semi-structured, or unstructured data. The clean separation of client and server at the HTTP web layer assures backward compatibility and better extensibility. Future work will focus on fully implementing the latest XML query facilities such as XQuery, XQuery Update, and XQuery Full Text to give the end-user state-of-the-art tools with which to query large-scale data sets. Especially the full-text feature will further improve the value of our infrastructure for the collaboration-oriented end-user because he or she can freely search in all comments and documents stored along with the spatio-temporal data. We also plan to investigate how to most efficiently distribute TreeTank for even better scalability.

6.3 Summary

We identified the data conversion step of the traditional workflow of Geographic Visual Analytics as a major bottleneck imposing large end-to-end delays between the query issuance and the retrieval of the final SVG ready for visualization. We suggest a new streamlined two-step workflow based on a native XML database promising better scalability and diminished delays while better supporting the interactivity aspect from the end-user perspective. The excellent compression ratio as well as the fast SVG delivery strongly encourage further research.

Both throughput and interactivity are essential for collaboration-oriented environments where end-users are used to interact in an asynchronous as well as a synchronous fashion. The support for the evolutionary growth of tree (XML) data structures as well as the ability to store and query statistical and SVG data, side-by-side, help to reduce unnecessary media breaks which hinder the dissemination of (visually) discovered knowledge.

Chapter 7

Conclusions

In the present thesis, we show that the adherence to the current *anti-evolutionary* approach mainly dictated by past technological and financial limitations deprives us from elegant yet well-performing applications providing far more insight into our data because they include past modifications and state. In addition, and to charm away any concerns, we show that our approach comes with the same linear scalability and logarithmic update properties as the existing one. However, we must change our minds and let the natural modification-driven *evolutionary* approach *consistently* pervade *all* of our concepts and thoughts.

7.1 Contributions

In Chapter 1, we have listed a number of claims to be addressed in this thesis. Now it is time to revisit them. Below, you will find each of these claims assessed.

1. **Degree of Granularity**
   We show that faster random-access storage hardware allows for an ever smaller granularity of the stored data. As such, storing the evolution of our data including even the tiniest of the intermediate steps gets ridiculously cheap.
   A lot of research currently tries to optimize file systems for flash-based or hybrid flash-, mechanical-, and cloud-based storage [AMS+13, KsKMP13, Ede11, JBLF10, CEKW08]. However, all this research still respects the file-level granularity. In contrast, we believe that the file-level granularity must be broken up to store and access even more fine-granular data structures. While we focus on the storage and versioning concepts, Alexander Holupirek showed that the applications can greatly benefit from direct access to fine-granular sub-file-level content [Hol06, Hol12]. This contribution was published as *Growing Persistent Trees into the 21st Century* [Kra08a].

2. **Evolution of State**
   We show that mandatory versioning, i.e., storing the modifications transforming one state into the other beautifully simplifies the applications and lays a solid foundation for backing up, distributing, and scaling of a data storage in a time- and resource-efficient fashion.
   Concentrating on the current state only leads to a continuous need to compare

states, i.e., find out, what changed between these two states. This is time-consuming, non-trivial and often left to manual work. Interestingly, more and more file systems introduce event-driven approaches [Ars07a] to overcome this overhead because they realize that a modification usually is tiny when compared to the whole state of the data. Notably, also database replication and clustering is mainly based on event or log streams [Pos11]. However, these event-driven approaches are not thought to the end: they are simply used to incrementally transfer the latest state and overwrite past states. We show that flash-based or future MRAM-based [ZZR+12] storage allows at the same time to store the modification history and the state. This contribution was also published in *Growing Persistent Trees into the 21st Century* [Kra08a].

3. ***Pre/Post* Tree Encoding with Logarithmic Update Complexity**
   We show that the pre/post tree encoding can be updated with logarithmic complexity $O(\log n)$ by using counted B+ trees. This is a significant improvement over the current $O(n)$ update complexity.
   The *pre/post* tree encoding allows for elegant and efficient querying [Gru02, GvKT03] at the cost of making updates prohibitive because of their $O(n)$ complexity. Because we strongly believe in the need for frequent modifications as found in any OLTP system, we tried to optimize the *pre/post* tree encoding for updates. First, we tried Skip Lists [Pug90] as a data structure instead of a read-only array. While Skip Lists hold their promise to be simple to implement, they proved to be a bit slower in practice by a small but constant block touch overhead when compared to B+ trees – even though the theoretical properties are the same as B+ trees. Second we tried counted B+ trees which allow positional access in $O(\log n)$ time [Tat04]. The counted B+ tree can be updated in any way indefinitely without loosing it's complexity properties. Peter Boncz achieved update capabilities, but by using the slightly adapted *pre/size/level* tree encoding [BMR05]. However, this update facility can not be updated indefinitely and must be reorganized at some point in time. Systems, such as MonetDB [Bon02], can greatly benefit from our contribution. This contribution was published as *Pushing XPath Accelerator to its Limits* [GHK+06].

4. **Linear Read Scalability beyond Memory Limitations**
   We show that the Parent/First Child/Left Sibling/Right Sibling tree encoding linearly scales beyond memory limitations when applied to persistent storage while keeping logarithmic update complexity. This allows to store and query tree-structured data sets orders of magnitudes bigger and faster than currently feasible.
   XML and the persistent storage of XML was ill-reputed to be slow and not scalable. Regardless of this, XML and its rich toolset are so widely used that it's hard to ignore it. Christan Grün showed that the *Pre/Dist/Size* tree encoding is linearly scalable beyond memory limitations and provides excellent properties for fast index-based queries [Grü10]. We showed that the *Parent/First Child/Left Sibling/Right Sibling* tree encoding also is linearly scalable beyond memory limitations and provides excellent properties for fast logarithmic updates. This contribution was published as *Pushing XPath Accelerator to its Limits* [GHK+06].

5. **Secure Node-Level Copy-on-Write**
   We show that the checksum-protected copy-on-write, a.k.a., the log-structured approach, can not only efficiently be applied to the file level but to the much finer-granular node level.
   File system research has shown that out-of-place (copy-on-write, log-structured)

updates can outperform in-place updates [RO92]. It has also been shown that journaling, a conceptually close relative to out-of-place updates, increases data availability [Jon08]. In addition, it has been shown that data security, i.e., integrity and encryption, compression, and de-duplication can be integrated with file systems in an elegant way [BM04]. In fact, these features proved to be so valuable, that they are now partly or fully integrated into existing file systems [Mic12]. We show that the same techniques can not only be used for file systems, but also at sub-file-level granularity, i.e., for node-level structured tree storage – with the same well-known performance, scalability, and space characteristics. This contribution was published as *Treetank, Designing A Versioned XML Storage* [GKW11] and awarded German patent number *DE 10 2008 024 809 B3* [Kra08b].

6. **Predictable Realtime Node-Level Access**
   We show that any past version, or the sequence of modifications resulting in that version, can be accessed at node level with constant, predictable costs satisfying realtime requirements. Current systems either have to store much more data to achieve this, either trade logarithmic read or write with linear read or write, or invest enormous computing resources.
   One of the design goals of SlidingSnapshot was its predictability. We guarantee, that any version of any node can be retrieved within $O(p \log n)$ I/O operations, where n denotes the total number of nodes in the system and p is the predefined number of pages to scan. To our knowledge, no other versioning system provides a facility to globally, and in advance, set p, i.e., they must scan a variable amount of pages depending on which node and which version to retrieve. In addition, we do not need a CPU-intensive algorithm to derive the node from a set of differential or incremental pages. This contribution is published here for the first time.

7. **Space-Efficient Node-Level Snapshot**
   We show that node-level snapshots consume less or at most the equal amount of space as page-level snapshots while still holding the predictability claim.
   Another of the design goals of SlidingSnapshot was its space-efficiency and is based on the following observation with modification-oriented workloads. If node-level snapshots must be taken at regular intervals to allow for predictable node retrieval (see contribution above), the snapshot of a node intermittently coincides with a modification on the same node and can be omitted. More modifications and larger node-level snapshot intervals therefore lead to more omitted node-level snapshots, i.e., space savings when compared to full page snapshots. In the worst case, i.e., all nodes are changed, it consumes as much space as the full page snapshot. In the best case, i.e., the set of changed nodes completely overlaps the set of snapshot candidate nodes, only c/m *th* of the space of the full page snapshot is used (c is the number of changed nodes, m the total number of nodes on the page). This contribution is published here for the first time.

8. **High-Level Language Block Access is Fast**
   We show that a high-level language implementation of a block-level protocol such as iSCSI can be on par or faster than a low-level language. This also benefits proof-of-concept implementations of new ideas because they can be done and evaluated faster.
   The initial intention for a high-level Java iSCSI implementation was the need for block-level access for our quick and dirty Java-based mockups and proof-of-concepts for TreeTank. Only later, we found that the high-level iSCSI implementation empowered us to test new ideas of how to distribute blocks to

different iSCSI targets [Lem08]. Eventually, Sebastian Graf et. al. confirmed our claim, that multi-threading is key to be on par or outperform native C-based iSCSI implementations [GBW09]. This contribution was published as *jSCSI – A Java iSCSI Initiator* [KWL+07].

9. **Temporal REST**
   We outline an elegant temporal extension to REST to generically access any version or past modification of a web resource.
   We observe that the requirement to access past versions of its data forces the involved application to reinvent a new API and implementation to achieve this. In addition, we see a strong shift towards web-based applications which facilitate deployment and find ideal conditions on now widespread mobile devices. Providing a generic, simple, and implementation-agnostic interface extension to a well-known interface such as REST promised to be the right way. In fact, it enabled us to extend an existing commercial web-based application with temporal features in virtually no time. This application provides access to past versions of auto-generated PDFs as well as fine-granular database records. Furthermore, Temporal REST inspired a JSON /XML server extension to OpenBaseSQL, a commercially available database server [Ope91]. Even though the versioning is just done with database triggers copying database records to another table upon modification, it still uses Temporal REST to expose these record versions. Future improvements to the storage implementation will therefore not break the interface and applications. This contribution was published as *Temporal REST—How to really exploit XML* [GK08].

10. **Improved Workflow for Geographic Visual Analytics**
    We show how to speed up interactive and collaborative applications in Geographic Visual Analytics by one third by eliminating a whole intermediate step.
    Our *evolutionary* tree-structured storage approach allows to skip the expensive step which converts geospatial data into SVG. By doing this, much more user requests can be handled per time unit, and the degree of interactivity is dramatically improved as the user actions are no longer a matter of minutes but seconds. Both throughput and interactivity are essential for collaboration-oriented environments where end-users are used to interact in an asynchronous as well as a synchronous fashion. The support for the evolutionary growth of tree (XML) data structures as well as the ability to store and query statistical and SVG data, side-by-side, help to reduce unnecessary media breaks which hinder the dissemination of (visually) discovered knowledge. This contribution was published as *Streamlined workflow for large-scale interactive geographic visual analytics* [KG08] and *An XML-based Infrastructure to Enhance Geographic Visual Analytics* [KGFW09].

7.2 Outlook

In this section, we present five ideas for future work that build upon our contributions. For most of them, the first steps have already be done. However, this work opens a wealth of opportunities, especially for interdisciplinary research. Most importantly, our work has been done around tree-structured XML as a mental gymnastic apparatus for this thesis. Nevertheless, our ideas can be expanded to other data structures and formats – a fact which on its own opens innumerable possibilities.

7.2.1 TreeTank Improvements

As TreeTank mainly is a log-structured system, we will, as future work, investigate, how to prune old versions to reclaim space. Further work will explore prefetching, caching, pipelining, and schema-aware techniques to exploit the Staircase Join-inherent knowledge about the XML data to minimize disk touches while maximizing CPU utilization [GvKT03]. Work has still to be done in extending the single write transaction semantics to multiple write transactions by applying, e.g., optimistic locking, or a clever data partitioning scheme using multiple instances of TreeTank. With respect to query performance, efficient query evaluation is key. The parallel and streamed execution of XPath and XQuery have been initially investigated by [Sch09] and [But07] respectively with promising results. In addition, the research on distributed TreeTank is already continued by Sebastian Graf et al. E.g., [Kra08c] shows, that TreeTank offers excellent performance when used as a storage node for large-scale network traffic analysis data sets in a peer-to-peer network. We believe that the distribution is key for performance and availability.

7.2.2 TreeTank Hardware

The specification of TreeTank allows an implementation based on well-known code (data structure, compression, hashing, encryption). There is a long track of efforts which aim at speeding up frequently used code by offloading it from general-purpose hardware to special ASICs (hardware can not be changed after deployment) or FPGAs (hardware can be changed after deployment) [Fi08, Sum08, Bro13, Ora13]. We implemented a proof-of-concept and offloaded the encryption code to special-purpose hardware. The setup consisted of an OpenBSD kernel routine called from TreeTank and then executed on Hi/fn 7954 security accelerator chip [Hif03]. Even in this simple setup, immediate improvements of 25% where possible. Offloading the whole security and compression code, maybe even the whole TreeTank into hardware, close to the flash storage, seems to be very promising with respect to execution performance and low energy consumption.

7.2.3 Evolutionary Indices

We showed how to provide predictable realtime node-level access to as well as space-efficient node-level snapshot for the main data structures. It is a well-known fact that any real-world data structure will need supporting indices to speed-up queries. In fact, one can tailor the optimal index for any specific query. Besides the traditional indices, full-text indices are now commodity. Markus Majer made an initial attempt to answer the question if versioned full-text indices will benefit from our contributions [Maj08]. He not only showed that this is perfectly reasonable but also that novel pruning and optimizing algorithms can be applied due to the fact that the modification history is available.

7.2.4 Evolutionary Applications

We showed that applications can greatly benefit from a generic versioned node-level storage. First, they are less complex to implement. Second, they are faster because intermediate layers can be eliminated. We implemented proof-of-concepts for five different applications which should provide an end-user interface to versioned data.

First, a tree editor to directly modify a TreeTank. Second, a text editor to edit documents in the *Open Document Format* [All02] stored in TreeTank. Third, a collaborative text editor to allow a team of authors to contribute different parts to a large document, also stored in TreeTank. Fourth, a semantic tree navigation application [Sch06]. Fifth, an XML mail collaboration application [Pet07]. In the course of this implementations, we realized, that there are three interesting directions when we immerse our versioning approach into the applications themselves.

Evolutionary Application Code: The collaborative text editor was coincidentally written in Adobe Flex [Ado04], which itself is encoded as XML. What initially was just a funny try, quickly proved to be a very powerful approach: We stored the Adobe Flex code itself in TreeTank besides the actual data. As a result, we could run any past version of our code against the data at that version. Not only do we get a simple version control system, but also the ability to access past versions of our data with the matching code – a fact that will be invaluable for long term archiving in the context of long-term code emulation [Wel08, OJD09].

Evolutionary Application Data: Current operating systems start to provide simple end-user access to versioned file-level data [BM04, Ars07b]. This got as easy as dragging a time slider widget in the frontend to quickly see which files have been created and when. Our tree and text editor applications extend this beyond the file-level and provides a time slider to easily slide through all versions of the tree or document and see who changed what. This proved very helpful for larger teams editing a document. At all times, everyone could see what changed between consecutively released versions. Not surprisingly, related work such as Etherpad [GIZ08] also experiments with a comparable end-user experience and shows that there is a strong demand for versioned, interactive, and collaborative, i.e., *evolutionary* access to application data.

Evolutionary Web Browsing: An idea we did never implement ourselves but is an obvious next step, is to write a time slider plugin for a temporal web browser. This would allow to browse websites and resources in the current but also any past state. For historians, e.g., it would be very interesting to browse news sites as they looked like years ago.

7.2.5 Evolutionary Schemas

A great part of todays engineering involves the definition of data schemas and then the implementation of the code to access that data. This is due to the object-relational impedance mismatch: the code is object-oriented, the data relational. The process is, first, to change the schema and, second, to let semi-automatic tools figure out the changes between the old and the new schema to generate the code required to adapt the data store. It turns out, that some schema changes need manual interaction because the schema modification code is too complex. As such, the current engineering process is *anti-evolutionary*. An *evolutionary* approach to the same engineering problem would be, first, to implement the modification to the schema, and, second, to automatically let the software generate the new schema both in the object-relational mapping (ORM) and the relational database. This would not only assure a full and automatic schema generation coverage, but also precise and loss-less documentation of the whole schema evolution. We believe that our suggested approach to schema handling and evolution has the potential to radically simplify ORM-based software development.

Bibliography

[AA99] Gennady L. Andrienko and Natalia V. Andrienko. Interactive maps for visual data exploration. *International Journal of Geographical Information Science*, 13(4):74–355, 1999.

[AAD+08] Gennady L. Andrienko, Natalia V. Andrienko, Jason Dykes, Sara I. Fabrikant, and Monica Wachowicz. Geovisualization of dynamics, movement and change: Key issues and developing approaches in visualization research. *Information Visualization*, 7(3):80–173, 2008.

[ABC+03] Serge Abiteboul, Angela Bonifati, Grégory Cobéna, Ioana Manolescu, and Tova Milo, editors. *Dynamic XML Documents with Distribution and Replication*, volume pp. 527-538, 2003.

[AD08] A-DATA. Vitesta Extreme Edition. http://techgage.com/article/a-data_2gb_pc2-8000_vitesta_extreme_edition/, 2008.

[Ada03] Adaptec. The Cost Benefits of an iSCSI SAN. http://www.adaptec.com/en-US/_whitepapers/tech, 2003.

[Ado99] Adobe. SVG Viewer. http://www.adobe.com/svg/, 1999.

[Ado04] Adobe. Flex. http://www.adobe.com/ch_de/products/flex.html, 2004.

[All02] ODF Alliance. Open Document Format. http://docs.oasis-open.org/office/v1.0/OpenDocument-v1.0-os.pdf, 2002.

[AMS+13] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, Francois Labelle, Nathan Coehlo, Xudong Shi, and Eric Schrock. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *Proceedings of the USENIX Annual Technical Conference*, pages 91–102, 2560 Ninth Street, Suite 215, Berkeley, CA 94710, USA, 2013.

[Apa97] Apache. Lucene. http://lucene.apache.org/, 1997.

[Apa99] Apache. Ant. http://ant.apache.org, 1999.

[Apa00] Apache. Subversion. http://subversion.apache.org, 2000.

[Apa05] Apache. Batik SVG Toolkit. http://xmlgraphics.apache.org/batik/, 2005.

[App07] Apple. Time Machine. A giant leap backward. http://www.apple.com/chde/support/timemachine/, 2007.

[Ars07a] ArsTechnica. FSEvents. http://arstechnica.com/reviews/os/mac-os-x-10-5.ars/7#fsevents, 2007.

[Ars07b] ArsTechnica. Time Machine. http://arstechnica.com/reviews/os/mac-os-x-10-5.ars/14#time-machine, 2007.

[AYBB+06] Sihem Amer-Yahia, Chavdar Botev, Stephen Buxton, Pat Case, Jochen Doerre, Darin McBeath, Michael Rys, and Jayavel Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text. Technical report, World Wide Web Consortium, 2006.

[BAH+03] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. USENIX Conference on File and Storage Technologies (FAST), 2003.

[Bar90] Richard Barker. *CASE Method: Entity Relationship Modelling*. Reading, MA: Addison-Wesley Professional, Tarrytown, New York, 1990.

[BC07] Angela Bonifati and Alfredo Cuzzocrea. Efficient Fragmentation of Large XML Documents. *LECTURE NOTES IN COMPUTER SCIENCE*, 4653(539), 2007.

[BCFK06] Peter Buneman, Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis, editors. *Using Partial Evaluation in Distributed Query Evaluation*, volume pp. 211-222, 2006.

[Ber91] Berkeley DB. http://www.oracle.com/technetwork/database/berkeleydb/overview, 1991.

[Ber03] Berkeley DB XML. http://www.oracle.com/technetwork/database/berkeleydb/overview/xquery-160889.html, 2003.

[BF05] Sujoe Bose and Leonidas Fegaras, editors. *XFrag: A Query Processing Framework for Fragmented XML Data*, number Proceedings of the WebDB, 2005.

[BG03] Jan-Marco Bremer and Michael Gertz, editors. *On Distributing XML Repositories*, number Proc. of WebDB, 2003.

[BGea06] Peter A. Boncz, Torsten Grust, and et al. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of ACM SIGMOD/PODS Int'l Conference on Management of Data/Principles of Database Systems*, Chicago, Illinois, USA, 2006.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of ACM SIGMOD/PODS Int'l Conference on Management of Data/Principles of Database Systems*, pages 310–321, Wisconsin, USA, June 2002.

[BM04] Jeff Bonwick and Bill Moore. ZFS: The last word in file systems. http://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf, accessed 10 November 2008 2004.

[BMR05] Peter A. Boncz, Stefan Manegold, and Jan Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery. In *XIME-P*, 2005.

[Bon02] Peter A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[BR01] Timo Boehme and Erhard Rahm. XMach-1: A Benchmark for XML Data Management. In *BTW*, 2001.

[Bro13] Broadcom. iSCSI Offload for 10GBE Converged Network Adapters. Technical report, 2013.

[But07] Daniel Butnaru. A Streamed Tuple Based XQuery Algebra Execution Engine. Bachelor's thesis, Universität Konstanz, 2007.

[CAE+02] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead. Versioning Extensions to WebDAV. RFC 3253, 2002.

[CEKW08] Henry Cook, Jonathan Ellithorpe, Laura Keys, and Andrew Waterman. IotaFS: Exploring File System Optimizations for SSDs. Technical report, Computer Science Division, University of California at Berkeley, 2008.

[CNP82] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti, editors. *Horizontal Data Partitioning in Database Design*, volume pp. 128-136, 1982.

[Con05] Noelios Consulting. Lightweight REST framework for Java. http://www.restlet.org, 2005.

[CP06] Yoon-Seop Chang and Hyeong-Dong Park. XML Web Service-based Development Model for Internet GIS Applications. *International Journal of Geographical Information Science*, 20(4):99–371, 2006.

[CTZ00] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Version Management of XML Documents. The World Wide Web and Databases, 2000.

[CVS89] CVS. http://savannah.nongnu.org/projects/cvs, 1989.

[DGB06] Robert I. Dunfey, Bruce M. Gittings, and James K. Batcheller. Towards an Open Architecture for Vector GIS. *Computers and Geosciences*, 32:32–1720, 2006.

[Dil08] Matthew Dillon. The Hammer Filesystem. http://www.dragonflybsd.org/hammer/hammer.pdf, 2008.

[dlB59] René de la Briandais. File Searching Using Variable Length Keys. In *Proceedings of Western Joint Computing Conference*, pages 295–298, 1959.

[DT05] Aaron Dailey and Scott Tracy. Using ISCSI Multipathing in the Solaris 10 Operating System. SUN BluePrints TM OnLine, December 2005.

[Ecl01] Eclipse. http://www.eclipse.org, 2001.

[Ede11] Nate Edel. MRAMFS: A Compressing File System for Byte-Addressable Non-Volatile RAM. Technical Report UCSC-SSRC-11-02, University of California, Santa Cruz, March 2011.

[FAA+01] David Fairbairn, Gennady L. Andrienko, Natalia V. Andrienko, Gerd Buziek, and Jason Dykes. Representation and its Relationship with Cartographic Visualization: a Research Agenda. *Cartography and Geographic Information Science*, 28(1), 2001.

[Fed01] Federal Information Processing Standards Publication 197. *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, November 2001.

[Fed02a] Federal Information Processing Standards Publication 180-2. *Secure Hash Standard*. National Institute of Standards and Technology, August 2002.

[Fed02b] Federal Information Processing Standards Publication 198. *The Keyed-Hash Message Authentication Code (HMAC)*. National Institute of Standards and Technology, March 2002.

[FFKZ10] Ghislain Fourny, Daniela Florescu, Donald Kossmann, and Markus Zacharioudakis. A Time Machine for XML: PUL Composition. XML Prague Conference, 2010.

[Fi08] Fusion-io. ioDrive. http://www.fusionio.com/products/iodrive2/, 2008.

[Fie00] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, California, 2000.

[FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley Publishing, 2003.

[FSC+03] Mary Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing XQuery 1.0: The Galax Experience. In *Proc. of Int'l Conference on Very Large Data Bases (VLDB)*, pages 1077–1080, Berlin, Germany, 2003.

[Gah07] Mark Gahegan. *Handbook of Geographic Information Science*, volume chapter Multivariate Geovisualization. Blackwell Publishers, 2007.

[GB09] Erich Gamma and Kent Beck. JUnit – A Regression Testing Framework. http://www.junit.org, 2009.

[GBW09] Sebastian Graf, Patrice Brend'amour, and Marcel Waldvogel. jSCSI 2.0 : Multithreaded Low-Level Distributed Block Access. Technical report, Universität Konstanz, 2009.

[GHK+06] Christian Grün, Alexander Holupirek, Marc Kramis, Marc H. Scholl, and Marcel Waldvogel. Pushing XPath Accelerator to its Limits. In *Proceedings of the First International Workshop on Performance and Evaluation of Data Management Systems (EXPDB 2006)*. ACM, 2006.

[GHS07] Christian Grün, Alexander Holupirek, and Marc H. Scholl. Visually Exploring and Querying XML with BaseX. In *12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW 2007)*, Aachen, Germany, March 2007. (Demo).

[Gia09] Georgios Giannakaras. Design and Evaluation of XML-based Geographical Visualization. Master's thesis, Universität Konstanz, 2009.

[GIZ08] David Greenspan, Aaron Iba, and J.D. Zamfirescu. EtherPad. http://etherpad.org, 2008.

[GK08] Georgios Giannakaras and Marc Kramis. Temporal REST—How to really exploit XML. Freiburg, Germany, 2008. IADIS International Conference WWW/Internet.

[GKW08] Sebastian Graf, Marc Kramis, and Marcel Waldvogel. Distributing XML with Focus on Parallel Evaluation. In *Sixth International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2008)*, pages 55–67, 2008.

[GKW11] Sebastian Graf, Marc Kramis, and Marcel Waldvogel. Treetank, Designing A Versioned XML Storage. In *XML Prague '11*, 2011.

[GM05] Binny S. Gill and Dharmendra S. Modha. WOW: Wise Ordering for Writes - Combining Spatial and Temporal Locality in Non-Volatile Caches. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, USA, Dec. 2005.

[GML98] GML. OpenGIS geography markup language (GML) encoding standard. http://www.opengeospatial.org/standards/gml, 1998.

[GNU86] GNUPlot. http://www.gnuplot.info, 1986.

[GR93] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, 1993.

[Gra14] Sebastian Graf. *Flexible Secure Cloud Storage*. PhD thesis, Universität Konstanz, 2014.

[Gro04] Object Management Group. Common Object Request Broker Architecture: Core Specification. http://www.omg.org/spec/, 2004.

[Gru86] Dick Grune. Concurrent Versions System, A Method for Independent Cooperation. Technical report, IR 113, Vrije Universiteit, 1986.

[Gru02] Torsten Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD international conference on management of data*, volume pp. 109-120, Madison, Wisconsin, 2002.

[Grü10] Christian Grün. *Storing and Querying Large XML Instances*. PhD thesis, Universität Konstanz, 2010.

[GvKT03] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *VLDB*, pages 524–525, 2003.

[GW08] Sebastian Graf and Marcel Waldvogel. Splitting and distributing large document-centric xml databases. Technical report kn-15-09-2008-disy-04, Universität Konstanz, 2008.

[GWF+99] Y. Goland, E. Whitehead, A. Faizi, S. R. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring – WebDAV. RFC 2518, 1999.

[Han03] David Heinemeier Hansson. Web development that doesn't hurt. http://www.rubyonrails.com/, 2003.

[Hif03] Hifn. Hi/fn 7954 Security Accelerator Chip. http://soekris.com/products/vpn14x1.html, 2003.

[HMF99] Gerald Huck, Ingo Macherius, and Peter Fankhauser. PDOM: Lightweight Persistency Support for the Document Object Model. German National Research Center for Information Technology, Integrated Publication and Information Systems Institute, 1999.

[Hol06] Alexander Holupirek. Implementing File Systems by XML-aware Databases. Master's thesis, Universität Konstanz, 2006.

[Hol12] Alexander Holupirek. *Declarative Access to Filesystem Data : New application domains for XML database management systems.* PhD thesis, Universität Konstanz, 2012.

[IBM55] IBM. 305 RAMAC. http://en.wikipedia.org/wiki/IBM_305, 1955.

[IBM70] IBM. DB2 Product Family. http://www-306.ibm.com/software/data/db2/, 1970.

[IBM07] IBM. Business Process Execution Language for Web Services version 1.1. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.

[IPe99] IPerf. University of Illinois. http://sourceforge.net/projects/iperf, 1999.

[iSC04] iSCSITarget. The iSCSI Enterprise Target Project. http://iscsitarget.sourceforge.net, 2004.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley-Interscience, New York, NY, April 1991.

[JBe01] JBench. http://www.yoda.arachsys.com/java/jbench, 2001.

[JBe03] JBenchmark. http://www.jbenchmark.com, 2003.

[JBLF10] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: A file System for Virtualized Flash Storage. In *In FAST10: Proc. of the Eighth USENIX Conf. on File and Storage Technologies (2010), USENIX Association*, 2010.

[JDTZ05] Song Jiang, Xiaoning Ding, Enhua Tan, and Xiaodong Zhang. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, USA, Dec. 2005.

[JKK+06] Halldór Janetzko, Daniel A. Keim, Marc Kramis, Florian Mansmann, and Marcel Waldvogel. Interactive Poster: Exploring Block Access Patterns of Native XML Storage. In *Proceedings of InfoVis 2006*, 2006.

[Jon08] Tim Jones. Anatomy of Linux journaling file systems. Technical report, IBM corporation, 2008.

[JPe07] JPerf. http://code.google.com/p/xjperf/, 2007.

[JPr01] JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html, 2001.

[JUn00] JUnit. http://www.junit.org, 2000.

[JUn01] JUnitPerf. http://clarkware.com/software/JUnitPerf.html, 2001.

[Kay98] Michael Kay. SAXON – The XSLT and XQuery Processor. http://saxon.sourceforge.net/, 1998.

[KG06a] Marc Kramis and Sebastian Graf. jSCSI – A Plain Java iSCSI Framework. http://jscsi.org/, 2006.

[KG06b] Marc Kramis and Sebastian Graf. Perfidix – No Discussions. Just Facts. http://perfidix.org/, 2006.

[KG08] Marc Kramis and Cedric Gabathuler. Streamlined workflow for large-scale interactive geographic visual analytics. GeoSpatial Visual Analytics, a Workshop at the GIScience 2008 Conference, 2008.

[KGFW09] Marc Kramis, Cedric Gabathuler, Sara Irina Fabrikant, and Marcel Waldvogel. An XML-based Infrastructure to Enhance Geographic Visual Analytics. *Cartography and Geographic Information Science*, 36(3):281–293, 2009.

[KKA95] Daniel A. Keim, Hans-Peter Kriegel, and Mihael Ankerst. Recursive Pattern: A Technique for Visualizing Very Large Amounts of Data. In *Proceedings of Sixth IEEE Visualization 1995 (VIS'95)*, 1995.

[KMSZ06] Daniel A. Keim, Florian Mansmann, Jörn Schneidewind, and Hartmut Ziegler. Challenges in Visual Data Analysis. *Information Visualization, Tenth International Conference on Information Visualisation, London, England*, IV:9–16, 2006.

[KOG07] Marc Kramis, Alexander Onea, and Sebastian Graf. PERFIDIX : a Generic Java Benchmarking Tool. Jazoon 2007 - The International Conference on Java Technology, 2007.

[Kra98] Menno-Jan Kraak. The Cartographic Visualization Process: From Presentation to Exploration. *Cartographic Journal*, 35, 1998.

[Kra08a] Marc Kramis. Growing Persistent Trees into the 21st Century. Technical report, Universität Konstanz, 2008.

[Kra08b] Marc Kramis. Verfahren zur Speicherung einer Mehrzahl von Revisionen von baumstrukturartig verknüpften Datenfamilienteilen, patent number DE 10 2008 024 809 B3, 2008.

[Kra08c] Thierry Kramis. DIPStorage – Distributed Storage Strategies for IP Traffic Traces. Master's thesis, Universität Zürich, 2008.

[KsKMP13] Yangwook Kang, Yang suk Kee, Ethan L. Miller, and Chanik Park. Enabling Cost-effective Data Processing with Smart SSD. In *the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST 13)*, May 2013.

[KVK99] George Karypis and of Minnesota Vipin Kumar, A.H.P.C.R. Parallel multilevel k-way partitioning scheme for irregular graphs. *SIAM Review*, 41:278–300, 1999.

[KWL+07] Marc Kramis, Volker Wildi, Bastian Lemke, Sebastian Graf, Halldór Janetzko, and Marcel Waldvogel. jSCSI – A Java iSCSI Initiator. In *Paper for: Jazoon'07 – Internationale Konferenz für Java-Technologie*. Universität Konstanz, 2007.

[LCP06] Wei Lu, Kenneth Chiu, and Yinfei Pan. A Parallel Approach to XML Parsing. The 7th IEEE/ACM International Conference on Grid Computing, 2006.

[Lem08] Bastian Lemke. Degrees of Freedom in Distributed Storage. Bachelor's thesis, Universität Konstanz, 2008.

[Ley02a] Michael Ley. DBLP – Digital Bibliography & Library Project. http://www.informatik.uni-trier.de/~ley/db/, 2002.

[Ley02b] Michael Ley. The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 1–10, 2002.

[Lib06] Leonid Libkin. Logics for Unranked Trees: An Overview. *CoRR*, 2006.

[Lic07] Johannes Lichtenberger. Eine Speicherverbrauchsanalyse verschiedener Revisionierungsverfahren für Wikipedia. Bachelor's thesis, Universität Konstanz, 2007.

[LM07] Sang-Won Lee and Bongki Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 55–66, New York, NY, USA, 2007. ACM.

[Luc07] Lucene "Getting started". http://lucene.apache.org/core/2_9_4/gettingstarted.html, 2007.

[LZS+02] Kevin Lü, Yuanling Zhu, Wenjun Sun, Shouxun Lin, and Jianping Fan, editors. *Parallel Processing XML Documents*, number Proceedings of the International Database Engineering and Applications Symposium (IDEASÕ02), 2002.

[Mac06] Matt Mackall. Towards a better SCM: Revlog and Mercurial. *Ottawa Linux Symposium*, July 2006.

[Mac08] Joshua MacDonald. Xdelta. http://xdelta.org, 2008.

[Maj08] Markus Majer. Archive Searching using fully revisioned Full-Text Index. Master's thesis, Universität Konstanz, 2008.

[MBHM13] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazières. Replication, History, and Grafting in the Ori File System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 151–166, New York, NY, USA, 2013. ACM.

[MCM+06] Alan M. MacEachren, G. Cai, M. McNeese, R. Sharma, and S. Fuhrmann. GeoCollaboration Crisis Management: Designing Technologies to Meet Real-World Needs. In *In Proceedings of the 2006 international conference on Digital government research, volume 151 of ACM International Conference Proceeding Series*, volume 151, pp. 71-72, New York, 2006. ACM Press.

[MD11] Muthukumar Murugan and David. H.C. Du. Rejuvenator: A Static Wear Leveling Algorithm for NAND Flash Memory with Minimized Overhead. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 0:1–12, 2011.

[Mic89] Microsoft. Microsoft SQL Server. http://www.microsoft.com/sqlserver/2008/en/us/default.aspx, 1989.

[Mic07] Microsoft. Project Astoria Team Blog. http://blogs.msdn.com/astoriateam/, 2007.

[Mic12] Microsoft. Resilient File System. http://msdn.microsoft.com/en-us/library/windows/desktop/hh848060%28v=vs.85%29.aspx, 2012.

[MK01] Alan M. MacEachren and Menno-Jan Kraak. Research Challenges in Geovisualization. *Cartography and Geographic Information Science*, 28(1):3–12, 2001.

[MMS93] Mitchell Marcus, Marc A. Marcinkiewicz, and Beatrice Santorini. Building A Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19:313–330, 1993.

[Moc07] Xuan Moc. Analyse von Metadaten-Strukturen zur Speicherung von Bäumen. Bachelor's thesis, Universität Konstanz, 2007.

[MS03] Hui Ma and Klaus-Dieter Schewe, editors. *Fragmentation of XML documents*, volume pp. 200–214, Manaus, Brazil, 2003.

[MS05] Hui Ma and Klaus-Dieter Schewe, editors. *Heuristic Horizontal XML Fragmentation*, number Proc. of CAiSE, 2005.

[MT94] Alan M. MacEachren and D.R.F. Taylor. *Color Use Guidelines for Mapping and Visualization*, pages 47–123. Number Visualization in Modern Cartography. Cynthia A. Brewer, Tarrytown, New York, 1994.

[MW99] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proc. of Int'l Conference on Very Large Data Bases (VLDB)*, pages 315–326, Edinburgh, Scotland, UK, Sept. 1999.

[NCWD84] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9:680–710, 1984.

[Net96] Microsoft Developer Network. DCOM Technical Overview. http://technet.microsoft.com/en-us/library/cc722925.aspx, 1996.

[NIS01a] NIST Special Publication 800-38A. *Recommendation for Block Cipher Modes of Operation*. National Institute of Standards and Technology, 2001.

[NIS01b] NIST Special Publication 800-88. *Guidelines for Media Sanitization*. National Institute of Standards and Technology, 2001.

[NK07] Suman Nath and Aman Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 410–419, New York, NY, USA, 2007. ACM.

[NS05] Mark Nottingham and Robert Sayre. The Atom Syndication Format. RFC 4287, 2005.

[NvdL05] Matthias Nicola and Bert van der Linden. Native XML Support in DB2 Universal Database. In *Proc. of Int'l Conference on Very Large Data Bases (VLDB)*, pages 1164–1174, Trondheim, Norway, Aug. 2005.

[NW01] Andreas Neumann and Andréas M. Winter, editors. *Time for SVG-towards high-quality interactive web-maps*, volume pp. 62-2349, Beijing, China, 2001.

[NW09] Andreas Neumann and Andréas M. Winter. Vector-based Web Cartography: Enabler SVG. http://www.carto.net/papers/svg/index_e.shtml, 2009.

[OAS91] OASIS. DocBook Technical Committee Document Repository. http://www.oasis-open.org/docbook/, 1991.

[OGC97] OGC. Overview of OGC's interoperability program. http://portal.opengeospatial.org/files/?artifact_id=6196, 1997.

[OGC02] OGC. Web Feature Service Implementation Specification. http://portal.opengeospatial.org/files/?artifact_id=8339, 2002.

[OJD09] Andrea Oermann, Gerald Jäschke, and Jana Dittmann. Vertrauenswürdige und abgesicherte Langzeitarchivierung multimedialer Inhalte. Technical report, Otto-von-Guericke-Universität Magdeburg, 2009.

[OMFB02] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. *Lecture Notes In Computer Science*, pages 109–127, 2002.

[OOP+04] 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.

[Ope91] OpenBase. JSON / XML Server. http://www.openbase.com, 1991.

[Ope99] Open Source Initiative. The BSD 3-Clause License. http://opensource.org/licenses/BSD-3-Clause, 1999.

[Ope05] Open-iSCSI. http://open-iscsi.org, 2005.

[Ora79] Oracle. Oracle Database. http://www.oracle.com/us/products/database/overview/index.html, 1979.

[Ora08] Oracle. StorageTek T10000 Tape Drive. http://www.oracle.com/us/products/servers-storage/storage/tape-storage/overview/index.html, 2008.

[Ora13] Oracle. SPARC T5 Processor. http://www.oracle.com/us/corporate/innovation/sparc-t5-deep-dive/index.html, 2013.

[Pet07] Tim Petrowsky. Evaluation zum Einsatz von XML in zukünftigen Mailsystemen. Bachelor's thesis, Universität Konstanz, 2007.

[PMA40] LLC Pinnicle Management Associates. Pareto Principle. http://www.pinnicle.com/Articles/Pareto_Principle/pareto_principle.html, 1940.

[Pol05] PolePosition. The Open Source Database Benchmark. http://www.polepos.org, 2005.

[Pos11] PostgreSQL. Streaming Replication. http://wiki.postgresql.org/wiki/Streaming_Replication, 2011.

[Pro07] Java Community Process. The Content Repository API for Java Technology Specification 2.0 (JSR 283). http://jcp.org/en/jsr/detail?id=283, 2007.

[Pug90] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM*, 33(6):668–676, June 1990.

[PZ04] Zhong-Ren Peng and Chuanrong Zhang. The roles of geography markup language (GML), scalable vector graphics (SVG), and web feature service (WFS) specifications in the development of Internet geographic information systems (GIS). *Journal of Geographical Systems*, 6(2):95–116, 2004.

[RO92] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[Sch06] Hannes Schwarz. Kontext-basierte multidimensionale Navigation in semistrukturierten Daten. Bachelor's thesis, Universität Konstanz, 2006.

[Sch09] Tina Scherer. Parallelverarbeitung von XPath 2.0. Master's thesis, Universität Konstanz, 2009.

[Sea08] Seagate. Savvio 15K. http://www.seagate.com/docs/pdf/datasheet/disc/ds_savvio_15k.pdf, 2008.

[SH05] Ripduman Sohan and Steven Hand. A User-Level Approach To Network Attached Storage. In *In Proceedings of the The IEEE Conference on Local Computer Networks 30th Anniversary*, volume pages 108-114, 2005.

[Sin06] Amit Singh. *Mac OS X Internals*. Addison-Wesley Professional, 2006.

[SMS+04] J. Stran, K. Meth, C. Sapuntzakis, E. Zeidner, and M. Chadalapaka. Internet Small Computer Systems Interface (iSCSI). RFC 3720, April 2004.

[Suc02] Dan Suciu. Distributed Query Evaluation on Semistructured Data. *ACM Transactions on Database Systems*, 27:1–62, 2002.

[Sum08] Tony Summers. Hardware based GZIP Compression, Benefits and Applications. Technical report, AHA Products Group, Comtech EF Data Corporation, 2008.

[Swi08] T. Swicegood. *Pragmatic Version Control: Using Git*. Pragmatic Bookshelf Series. Pragmatic Programmers, LLC, 2008.

[SWK+01] Albrecht Schmidt, Florian Waas, Martin Kersten, Daniela Florescu, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Why and How to Benchmark XML Databases. *ACM SIGMOD Record*, 30:27–30, 2001.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. of Int'l Conference on Very Large Data Bases (VLDB)*, volume pp. 974–985, Hong Kong, China, Aug. 2002.

[Tat04] Simon Tatham. Counted B-Trees. http://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html, 2004.

[TPT05] TPTP. http://www.eclipse.org/tptp, 2005.

[W3C08] W3C. Efficient XML Enterchange Working Group. http://www.w3.org/XML/EXI/, 2008.

[W3C11] W3C. Coordinate Systems, Transformations and Units. http://www.w3.org/TR/SVG/coords.html, 2011.

[Wel08] Randolph Welte. *Funktionale Langzeitarchivierung digitaler Objekte*. PhD thesis, Albert-Ludwigs-Universität Freiburg im Breisgau, 2008.

[Wil07] Volker Wildi. Java iSCSI Initiator. Master's thesis, Universität Konstanz, 2007.

[Wor08] Worldbank. The Worldbank Data & Research. http://econ.worldbank.org, November 2008.

[WWA93] Randolph Wang, Olph Y. Wang, and Thomas E. Anderson. xFS: A Wide Area Mass Storage File System. Technical Report UCB/CSD-93-783, EECS Department, University of California, Berkeley, 1993.

[XH05a] X-Hive. Improve Operational Performance by Managing Your Complex Documentation with X-Hive. http://www.x-hive.com/, 2005.

[XH05b] X-Hive. X-Hive DB Version 7.2.2. http://www.xhive.com/, 2005.

[Xim04] XimpleWare. VTD-XML. http://vtd-xml.sourceforge.net/, 2004.

[YZ08] Xiaobai Yao and Liang Zou. Interoperable Internet Mapping: An Open Source Approach. *Cartography and Geographic Information Science*, 35(4):93–279, 2008.

[ZFS04] ZFS. ZFS Documentation. http://open-zfs.org/wiki/Documentation, 2004.

[ZK09] Vyacheslav Zholudev and Michael Kohlhase. TNTBase: Versioned Storage for XML. Balisage: The Markup Conference, 2009.

[ZZR+12] Y. Zhang, W. S. Zhao, D. Ravelosona, J. O. Klein, J. V. Kim, and C. Chappert. Perpendicular-Magnetic-Anisotropy CoFeB Racetrack Memory. 2012.