Enhancing XML Preservation and Workflows

Viacheslav Zholudev
School of Engineering and Science
Jacobs University Bremen

A thesis submitted for the degree of
Doctor of Philosophy in Computer Science

Dissertation Committee:
Prof. Dr. Michael Kohlhase (supervisor)
Prof. Dr. Peter Baumann
Prof. Dr. Dieter Hutter

Defended: 2012, July 17th
To my parents...
Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This thesis has not previously been presented in identical or similar form to any other German or foreign examination board.

Viacheslav Zholudev
Abstract

With the proliferation of computers and networked resources the amount of data available on personal computers and on the Web is growing exponentially. Handling of data becomes more complex as the scale expands. Moreover, data collections change in place over time, so that one of the important challenges consists in supporting a life-cycle of data. In particular, relevant parts of information have to be preserved and made accessible for future retrieval.

XML is a wide-spread language for encoding documents that is both machine-readable and human-readable. It plays a unique role on the Web and in other areas of digital life: publishers, libraries, warehouses, technical writers, to name a few, – all make extensive use of XML for encoding books, articles, product information, documentation, interchanging and transformation of data, and the extraction and aggregation of relevant pieces of information. Contemporary web services are stepping away from traditional relational data models and are moving towards semi-structured data representations and utilization of NoSQL, and, in particular, XML databases for persisting data. A considerable technological stack has been built around XML, making it even stronger as a format.

The development of XML technologies specifications is a never-ending process and emerging implementations of them push the progress further. However, there are still many open problems and challenges to be addressed in the XML domain. This thesis selects a number of the unsolved ones and suggests solutions; concretely (i) Versioning support for XML, (ii) XML databases views as counterparts of relational database views, (iii) Support for XML document templating.

Solving the XML versioning problem for a subset of use cases enables an XML persistence layer that tracks the history of changes and provides XML database functionality like querying or indexing of data. The XML database views concepts allows to abstract away from the notion of XML files and think in terms of customizable and editable abstract XML entities with origins in some XML documents. Finally, support for XML document templating facilitates separation of responsibilities while authoring XML documents. Being able to use diverse expertise of developers in different phases of document creation, in turn, optimizes the whole authoring workflow.

Highlighting the practical value of this work, all the concepts described in this thesis are implemented and integrated into the TNTBase system (http://tntbase.org/). Over the last 4 years TNTBase has been constantly utilized in a number of mainly research projects maturing by receiving feedback from its users. The main target of the TNTBase project was providing a versioned repository for the Open Mathematical Documents (OMDoc) with a strong focus on XML-related functionality. Since the
OMDoc format combines data-like aspects (axioms, theorems, examples, etc.) with document-like aspects (sections, paragraphs, etc.), the number of applications was diverse, and therefore all the approaches have been generalized to make it possible to adapt TNTBASE to other domains and XML languages. As a result, TNTBASE has been deployed in multiple real-life scenarios where it has been used on a daily basis.
Acknowledgements

A lot of people have helped me to accomplish results described in this thesis.

First of all I would like to thank my supervisors: my “doctor father” Michael Kohlhase has helped me in all aspects of doing a PhD research starting from elaborating my vision and ending with correcting my English writing. Peter Baumann and Dieter Hutter have considerably helped me with looking at my research from different perspectives and questioning research contributions that provoked more thoughts and deeper understanding of the subject.

All my colleagues from the KWARC group made a considerable contribution to the development of the TNTBase system. Andrea Kohlhase, Christine Müller, Normen Müller, Christoph Lange, Florian Rabe, Catalin David, Deyan Ginev, Dimitar Misev, Fulya Horozal, Constantin Jucovschi, Bogdan Matican shared their feedback with me and came up with great ideas for TNTBase case studies that have significantly shaped up the system and allowed me to understand what is still lacking in the XML domain. I want to thank other people with whom I had a chance to collaborate: Nikita Zhiltsov, Sebastian Graf, Alexander Holupirek, Christian Grün. Immanuel Normann, a former KWARC group member and John Batemann from University of Bremen provided an opportunity to test TNTBase in the completely new environment: in the ontology field.

Mihai Cirlanaru provided some TNTBase logos and pictures, whom I would like to thank as well.

Erik Hennum from MarkLogic Inc. initiated the development of the XSLT/XQuery tag libraries technology (Chapter 5) by contacting me after my talk on virtual documents at the Balisage 2010 conference. The work on tag libraries has been done in collaboration with him and used in this thesis with his permission.

Additionally, colleagues from DFKI, Serge Autexier, Dominik Dietrich and Dieter Hutter allowed me to extend the number of TNTBase case studies by experimenting with management of change for OMDoc documents.

Considerable amount of my work involved dealing with Berkeley DB XML, and the following people from Oracle helped me a lot by sharing the expertise in the XML and XML-databases field: George Feinberg, John Snelson and Lauren Foutz.

Additionally, I want to thank Andrea Kohlhase, Michael Kohlhase, Raul Cajias, Patrice Lopez, Faye Stringer, Christoph Lange, Otilia Dreglea, Niall Kelly for proof-reading my thesis.

Nikolay Melnikov, Christoph Lange and Catalin David substantially supported me in maintaining servers used for hosting TNTBase installations. substantially I received financial suport from Jacobs University Bremen, DFKI Bremen and the KWARC group, in particular, whom I would like to thank too.

Apart from the “scientific” list above I would like to express my gratitude to people who gave me inspiration and made the PhD path more pleasant: Otilia Dreglea, Nikolay Melnikov, Michael Ludwig, Lech Rzedzicki and the Bremen ice-hockey team “Bremer Brigade”.

i
# CONTENTS

## I Introduction

1 Introduction 3

1.1 Visions with XML: A Real World Use Case 4

1.2 Open Challenges in XML 7

1.3 Contribution of the Thesis 9

1.4 Structure of this Thesis 9

## II A Versioned XML Database

2 An XML-enabled Repository 13

2.1 Introduction 13

2.2 Fundamentals 14

2.2.1 Basic Concepts 14

2.2.2 VCS Concepts 16

2.2.3 Operations on a Repository 23

2.2.4 Relation between FS-trees and Repositories during a Repository Lifetime 31

2.2.5 Data Store Repository 34

2.3 XML-enabled Repository (XeR) 34

2.3.1 XPath, XQuery Data Model 34

2.3.2 Building an XML-enabled Repository 35

2.4 Towards An XML-Enabled Repository 36

2.4.1 Methodology and General Architecture 36

2.4.2 A Data Store Function 37

2.4.3 A Key-Mapping Function for Embedding $X$ into $V$ 38

2.5 The XeR Implementation – xSVN 39

2.5.1 What is an XML File for xSVN? 40

2.5.2 Integration 40

2.6 Conclusion 43

## 3 Towards a Versioned XML Database 45

3.1 Introduction 45

3.1.1 Goals 46
3.2 Versioned XML Database Fundamentals ........................................... 47
  3.2.1 XML-enabled Repository with Metadata ................................. 50
  3.2.2 FS-Aware Versioned XML Database ....................................... 51
  3.2.3 A File System for $X$ .......................................................... 52
  3.2.4 Summary .............................................................................. 55
3.3 A Versioned XML-Database Toolkit ................................................. 58
  3.3.1 An XQuery Library .................................................................. 59
  3.3.2 xAccessor and VcsAccesssor .................................................. 59
  3.3.3 A VXD Interface ..................................................................... 63
3.4 Open Problems and Directions to Solve Them ..................................... 63
3.5 Related Work .................................................................................. 63
  3.5.1 A Time Machine for XML ....................................................... 63
  3.5.2 xDB Versioning ...................................................................... 63
  3.5.3 DeepFS .................................................................................. 63
3.6 Implementation – TNTBase ............................................................... 64
  3.6.1 Consistency of TNTBase FS ..................................................... 65
3.7 Conclusion ...................................................................................... 66

III XML Workflows

4 Scripting XML Documents ................................................................. 71
  4.1 Introduction .................................................................................. 71
    4.1.1 Database Views ...................................................................... 72
  4.2 Virtual Document Fundamentals .................................................... 74
    4.2.1 Defining Virtual Documents .................................................. 75
    4.2.2 Virtual Documents by Example ............................................. 77
    4.2.3 VDoc Specifications and Operational Semantics ..................... 79
    4.2.4 VDocs as XML Database FS Entities ..................................... 83
    4.2.5 VDocs as XML Database Views ............................................ 85
    4.2.6 Editing Virtual Documents ................................................... 86
  4.3 Integrating with an XML Database ................................................. 94
    4.3.1 Embedding Node Origins ..................................................... 94
    4.3.2 Extending the VXD Toolkit .................................................. 94
    4.3.3 Versioned Snapshots and Materializing VDocs ....................... 95
    4.3.4 VDocs and Versioning ......................................................... 96
  4.4 Related Work ............................................................................... 97
    4.4.1 Drupal ................................................................................ 97
    4.4.2 Materialized XML Views .................................................... 97
  4.5 Implementation in TNTBase .......................................................... 98
  4.6 VDoc Use Cases .......................................................................... 99
    4.6.1 Automated Exam Generation .............................................. 99
    4.6.2 Multiple Versions of Documents ......................................... 100
    4.6.3 Managing Document Collections ....................................... 101
## CONTENTS

### 8 Performance and Stability
8.1 Introduction .................................................. 163
8.2 Comparing to SVN ............................................. 164
   8.2.1 Memory Usage in xSVN ............................. 165
8.3 Comparing to DB XML ....................................... 165
8.4 Other Tests .................................................. 167
   8.4.1 Concurrent Access ................................. 167
   8.4.2 Integrity Tests ................................. 168
8.5 Problems in TNTBase ...................................... 169
8.6 Conclusion .................................................. 171

### V Conclusion and Future Work

### 9 Conclusion and Future Work
9.1 Summary and Contributions ............................... 175
9.2 Future Work ................................................ 177

### VI Appendix

#### A Notations Overview and Abbreviations
   A.1 Notations Overview ..................................... 183
   A.2 Abbreviations ........................................... 185

#### B Building Bridges between Theory and Practice
   B.1 What Should be Considered as XML for an XeR ......... 187
   B.2 What Can Simplify Integration of an XML DB into a VCS 188
   B.3 Any XML Database is Metadata-enabled ............... 189
   B.4 XQuery Update with a Temporal Aspect ............... 189
   B.5 REST Interface for a VXD .......................... 190
   B.6 XQuery External Functions .......................... 191
   B.7 Why not XML-diff in the First Place ................ 191

#### C The TNTBase XQuery Library

### vi
Part I

Introduction
Chapter 1

INTRODUCTION

The difference between theory and practice is that in theory, there is no difference between theory and practice.

Richard Moore

With the proliferation of computers and networked resources the amount of data available on personal computers and on the Web is growing exponentially. Handling of data becomes more complex as the scale expands. Moreover, data collections change in place over the time, so that one of the important challenges consists in supporting a life-cycle of data. In particular, relevant parts of information have to be preserved and made accessible for future retrieval.

In order to do this, we must distinguish data itself from their presentation: For instance, a title and an album of a song stored on a file system stay the same while an operating system may use a different way to present those metadata to users in the future. Another example could be a presentation of a + sign that can mean addition of numbers or concatenation of strings, depending on the context \[KK06\]. XML \[Bra+08\] is a common such format for representing data, typically, semi-structured data, i.e., those that do not fit into a relational databases model. Its design goals are simplicity, generality, and usability over the Internet \[Bra+08\]. The success of XML as a format can be also attributed to its following properties:

- Strong support for heterogeneous data
- A culture of grammar-based validation
- Enough syntactic redundancy (end tags, quoted attribute values) to detect a lot of common errors
- Fairly simple to parse compared to SGML \[Cla97\] (no SGML declaration) or HTML \[RHJ98\] (no guesswork)
1. INTRODUCTION

- A ban of silent error correction lets XML be part of civil-engineering-level work
- Suitable for document transcriptions in most cases
- A culture of declarative approaches – easier to maintain than procedural approaches – amenable to optimization

Based on the active development of XML databases in the last decade we are seeing the growth of the deep Web for XML documents: Surface documents (what we see in our browsers) are assembled, aggregated and mashed up from background information persisted in XML. A solid infrastructure has been established around the XML domain by standards and their implementations to approach XML preservation, validation, fragment inclusion, transformation, querying, etc.

XML is ubiquitous and plays an important role in many aspects of the informational world. So far hundreds of XML-based formats have been developed \[Pag\]. Many publishing companies utilize XML and XML workflows to deliver content to readers \[Sku\]. XML also serves as a base language for multiple communication protocols, e.g. SOAP \[Gud+03\] or XML-RPC \[Scr10\]. Moreover, XML is also used as an interface language and a means for representing/interchanging data like XML \[Xmi\] or SVG \[JFF02\]. As a document markup language XML can represent narrative structure (DocBook \[WM08\]) or encoding semantic relations in mathematics (OMDoc \[Koh06b\]). Furthermore, XML-related workflows can be enhanced by technologies that utilize XML itself: XML validation (XML Schema Definition language \[GSMT09\], RELAX NG \[Rel\]), transformations (XSLT \[Kay07\]) and pipelines (XProc \[WMT10\]).

Considering its success, XML is still a young technology: it came to existence a bit earlier than a decade ago \[Xbi\]. Many other components of the XML technology stack are much younger. For instance, the first XQuery \[Boa+07\] W3C recommendation as well as the specification for XSLT 2.0 were released in the beginning of 2007. The development of XML-related specifications is an on-going work and therefore lots of open challenges are expected to be addressed from the XML domain. A technology is useful when its components are elaborated and well-integrated between each other. The ultimate goal of this thesis is to contribute to the XML technology stack, and thereby make the whole XML domain more powerful and easier to deal with.

To better understand the problems solved by this work, let us begin with a real world use case that shows why XML is important and introduces the reader to the problems that still need to be solved.

1.1 Visions with XML: A Real World Use Case

Assume that Joe is writing a next edition of the book on Algorithms and Data Structures and he uses some XML language to “encode” the book contents. He has already published several editions of his book, but small errata were found in the previous editions and therefore the corrections should be released online. The book materials consist of multiple XML documents with encoded relations between each other. The
types of these relations are e.g. the inclusion of one fragment into another, dependency links or cross-references. Obviously, Joe needs a toolset for (i) storing content in some data store, (ii) editing/managing content, and (iii) releasing/publishing content. All operations are interconnected and may involve an arbitrary level of complexity. The more tools are in the toolset, the higher the overall complexity of all workflows, and the more efforts are needed to support them. Efforts can be reduced if a big part of complexity is shifted towards a data store. The problem here is to have a unified toolset with data store capabilities that lighten the burden of Joe’s daily workflows. Let us justify what requirements/tasks Joe might have.

Content Management Maintaining several book editions naturally fits into the version control system (VCS) concept well-known from software engineering. The history aspect is appealing for managing several versions of the book: each edition would correspond to a particular branch. More advanced VCS features such as “automated merging” can be used for bringing errata from branches to the trunk. Ability to work simultaneously on the same files in contemporary version control systems enables collaboration with Joe’s friends who are experts in particular topics. Furthermore, a VCS-like system would also serve as an archiving tool preserving the whole development cycle in time: VCS capabilities can help to figure out unfortunate changes and to revert to the latest successful revision.

Since the book sources are in XML, an underlying data store could generate more informative differences between revisions by taking into account the XML structure. More informative differences mean more control over changes, and therefore lead to better management of the document collection. For example, a change could tell Joe that a particular title has been changed instead of informing him about the affected line number in the document.

Content Querying An electronic book provides an opportunity to enable additional services by aggregating and presenting information encoded in XML. Aggregation becomes a mechanical exercise when a toolset supports some querying language. XQuery is such a standard for XML. It is powerful enough to perform arbitrary complex queries, performance of which is only bound by the amount of data and a query processor implementation. Example of possible queries include retrieving all exercises from the book based on a certain topic, or gathering some statistics about the book such as the number of topics, search-related theorems, or the like. Querying across document history might also be beneficial. As an example, Joe may want to obtain all exercises that were introduced in the latest book revision but did not exist before.

Content Validation and Presentation One of the beneficial parts of XML is an existing stack of technologies for validating and transforming XML content. For instance, XML Schema or RELAX NG for validation and XSLT for transformation. Support of those on the data store level simplifies the toolset and removes complexity of integrating different tools into the Joe’s workflow. Speaking in
terms of a VCS, validation upon commit might be crucial for supporting integrity of data in the data store and transforming book sources into HTML in the post-commit phase may help Joe to preview the outcome of his work. This, basically, means a faster feedback loop and better control over the changes. More advanced transformation workflows behind XSLT capabilities can bring additional value to Joe. For instance, transforming a book into a graph of dependent concepts in the material would, certainly, help to maintain the book’s coherence by abstracting away from the XML syntax.

Content Customization Apart from straightforward query aggregations, more advanced use cases may appear in practice. Assume that Joe wants to have a special “view” on the content that contains all coding exercises from the book on algorithms of a certain topic and of certain difficulty. This can be used as a teaser on the web-page. To give an impression that books covers many topics from simple to the advanced ones, Joe envisions a possibility for a web-page visitors to choose a level of complexity of the provided set of exercises. Since exercises in the scenario with Joe are changed more frequently than other parts of the book, typos are more probable in those exercises. Therefore it would be beneficial if corrections could be made directly from the exercises view. Modifying content in such “views” would result in changes in the original XML sources in the underlying data store. This scenario removes another level of complexity by abstracting away from the notion of files, and replacing it by operations on the book entities (exercises, in this example).

Agile Documents Additionally to the views exemplified above, Joe may want to produce agile (X)HTML documents that contain statistics about the book: number of exercises per topic, number of pages per chapter, list of topics topologically sorted, etc. While Joe is not an expert in XSLT or XQuery which are needed to realize that task, he is experienced in styling and presenting information to an end user. Therefore, it would be beneficial if Bob, a good friend of Joe and an experienced XQuery programmer, could write data retrieval logic in the form of isolated units, and Joe could reuse them by embedding the code to a target template. Support for such document “scripting” in Joe’s toolset can optimize certain routine tasks and facilitate reusability of ad-hoc data retrieval units.

Real World Properties Last but not least, a system that Joe would want to use needs to be scalable, efficient and reliable. Only then such a system can be used in real world situations.

As we have seen from above, such workflow support for XML documents makes sense. In order to make them all possible we need to solve the issues listed in the next section.
1.2 Open Challenges in XML

XML Versioning Data are constantly changing – e.g. it has been estimated that 20% of the surface Web changes daily and 30% monthly [CGM00; Fet+03]. While archiving services like the Wayback Machine try to get a grip on this for the surface web, we still do not have an elaborated infrastructure for managing the changes in the deep web for XML. Version Control Systems like CVS [Cvs] or Subversion [Apa] which have dramatically improved collaboration workflows in software engineering are deeply text-based (e.g. wrt. diff/patch/merge algorithms [Joh96]) and are not integrated very well with XML technologies. This makes use of classical VCSs not optimal candidates for storing XML and managing history of changes. On the other hand, some relational databases are capable of preserving history of persisted data [DDL02], but this does not seem to have counterparts in the XML domain (with a few relatively new exceptions, most notable of which are covered in Section 3.5).

In Section 1.1 we explained that versioning capabilities were important for the XML domain. Whereas versioning is not necessarily required in all projects, it naturally fits into some document development cycles that deal with XML and facilitates collaboration between contributors. However, non-XML content is often not less important and is managed together with XML documents. Therefore functionality of dealing with non-XML content should not be sacrificed to the XML domain, but rather complemented.

XML databases without versioning capabilities store full versions of XML documents for faster retrieval of XML nodes. Versioning of XML documents can be naively done by storing consecutive versions of XML documents as full text as opposed to the incremental approach where consecutive versions are stored as differences against the previous ones. The downside of this approach is high disk space consumption and inefficiency of copying operations where the identical content is duplicated in the data store. Although concerns about storage space significantly went down over the last years due to increasing hard drive sizes and decreasing of prices, it can still be an issue in some projects, for instance, in the publishing field. Even if just a small XML file of 5 MB is changed frequently by multiple people a day, the free storage space will diminish rapidly. If e.g. a file is changed and committed by 10 people 10 times a day, \(5 \times 10 \times 10 = 500\) MB a day of disk space will be consumed. That means ca. 15 GB a month. And if there are 10 files that are modified in such a way each day, it will mean that \(10 \times 15 \times 12 = 1.8\) TB of disk space will be required a year. This is too much especially considering that previous revisions are rarely accessed. It is a huge cost for arguably gained advantages of storing full versions of XML documents.

The author proposes a solution that uses the incremental approach that is space-efficient and allows to seamlessly enable XML-related functionality (see below). An additional value of this solution is a pragmatic approach of combining a VCS and an XML database. It requires less efforts than building a new system completely from scratch and does not sacrifice efficiency. Moreover, due to a big amount of implementations available for both classes of systems, the approach leaves enough freedom for...
1. INTRODUCTION

experiments and figuring out the technical combination that suits certain requirements. In order to achieve this the author came up with a rigorous theory to understand when and how it is possible (Chapter 2).

XML Querying Most XML data stores provide querying facilities via XQuery that enable XML data aggregation and retrieval. As we see from Section 1.1 querying content may be important for various scenarios. While the XML database querying is a well studied topic, it still had to be applied to the XML versioning approach introduced in this work. Chapter 3 solves this problem without losing expressiveness of the query language and efficiency of data retrieval. Furthermore, the author believes that the XML querying facilities should be aware of the file system structure of the underlying data store. The reason for it is that file systems exist for more than 40 years, and they shaped how users think when it comes to organizing data or items of knowledge. When putting a part of the file system tree into a data store it becomes extremely natural to reflect the file system structure in it. Thus, file systems paths become a natural way of addressing documents within the data store too. Hence, when working with a data store that contains parts of a file system tree, it is more practical to reflect its structure in the data store rather than reeducate users to deal with a different storage model. Therefore, it becomes a challenge to give users access to a document hierarchy structure when querying the content. Chapter 3 covers this as well.

XML Views As we have seen from the Section 1.1 views in the scope of XML databases can be a powerful tool. While the concept of views is widely adopted by most relational databases, in the XML domain only several attempts have been made recently. Bringing this concept to the XML database field could extend an XML stack of technologies and provide finer data granularity than XML documents, and hence more flexibility. Chapter 4 proposes a general solution to this problem as well as its implementation showcased by real world applications. Last but not least, applicability to a versioning data store that we have introduced in Chapters 2 and 3 is also discussed.

XML Authoring The agile documents item from previous section has showed an important aspect in XML authoring: enhancement of collaboration by separation of responsibilities. Looking at the number of standards to work with XML it gives us a nice opportunity to refine an XML toolkit and make XML authoring more efficient and painless for people with different levels of XML expertise. Chapter 5 presents a new technology that aims at solving problems that existing approaches are lacking.

Implementation Support Chapters 6, 7 and 8 show that the solution to the above problems is practically possible and efficient enough to be used in the real world scenarios.
1.3 Contribution of the Thesis

The high level goals of this work were:

(G1) Make XML technologies an efficient tool for long haul data preservation and history tracking.

(G2) Achieve more fine-grained and efficient separation between data and presentation.

(G3) Improve authoring of XML documents.

(G4) Enhance data interchange between XML-driven applications by supplying them with a powerful XML data store with rich interfaces.

The author has built a toolkit for preserving, maintaining and processing XML documents that enables feature-rich extensible and robust software systems for the XML domain. As a proof of concept the TNTBase system has been implemented that enables the features of the use case in Section 1.1.

The TNTBase system was initially motivated by the data store needs of the OM-Doc project (Open Mathematical Documents [Koh06b]), an XML-based representation format for structural mathematical knowledge. Many of the development requirements and ensuing problems of the XML community can be extrapolated from OMDoc-based applications and their data store needs since OMDoc combines data-like aspects (axioms, theorems, examples, etc.) with document-like aspects (sections, paragraphs, etc.), and therefore this extrapolation did not restrict generality of the development direction of TNTBase.

As we will see in Chapter 7, TNTBase has been successfully employed as a data store backend for different systems: a change management system DocTIP [Doc], a collaborative community-based reader Planetary [Koh+11; Dav+; Pla], an ontology data store BioPortal [BO09], etc. Any specializations to a particular use case has been built on top of the TNTBase core, making the main concepts, algorithms, engineering techniques of this work independent from a particular system or an XML language. The example from the OMDoc domain is thoroughly elaborated in Chapter 6 in an abstract way. Thus, the author envisions this chapter also as a guideline of employing TNTBase in a custom XML-enabled scenario.

1.4 Structure of this Thesis

This thesis is structured such that the chapters solve the problems mentioned above in the same order as already described. In Part II and III the author describes an approach for building a versioned XML data store and enhancing XML workflows, correspondingly. Part IV provides a practical extension of TNTBase to the OMDoc domain, TNTBase case studies as well as some performance measurements and system stability experiments.

Essentially, we will be gradually building the concept of the XML toolkit. The main part is the XML data store concept that support versioning. We start building
1. INTRODUCTION

its core in Chapter 2 and add XML-related interfaces in Chapter 3. Only then the discussed toolkit is reinforced by technologies that improve XML document workflows (Chapters 4 and 5).

In this thesis we use a considerable amount of notations and abbreviations. The full list of those can be found in Appendix A.1 and Appendix A.2.
Part II

A Versioned XML Database
Chapter 2

An XML-enabled Repository

If I had eight hours to chop down a tree, I’d spend six sharpening my axe.

Abraham Lincoln

2.1 Introduction

In order to approach open challenges described and exemplified in Sections 1.1 and 1.2 we have to start with the basics, gradually progressing based on the theoretical and practical results obtained before. In this chapter we will start with building a tool for archiving and retrieving pieces of semi-structured data in the form of XML. As discussed in the introduction, version control systems appeal to us due to convenience of collaborative editing and managing collections of documents. However, they do it without taking the document format into consideration. In particular, versioning of XML is a problem that has not been much looked into despite the fact that XML is nowadays ubiquitous on the Web. That drives the strong motivation to have the “XML-aware” versioned repositories. At the same time, a lot of work has been done with respect to XML-native databases that have an XML-specific data model in conjunction with XML fragment access by means of a query language and corresponding query processors.

That said, two classes of systems, namely XML databases and version control systems, are definitely useful for XML data, but reside on two different poles. What if we could come up with a framework that unites functionality needed for XML?

The goal of this chapter is to show that it is indeed possible. We will begin with working out a rigorous mathematical model of classical versioning and will extend it towards the XML domain to get a foundation for XML versioning. Furthermore, we propose engineering techniques that demonstrate how the model can be employed in practical applications. Throughout this chapter these engineering techniques will be justified, thoroughly analyzed and exemplified on our implementation – the TNTBASE system.
2. AN XML-ENABLED REPOSITORY

Although the theoretical concepts described here are independent of any existing technologies, the contributed engineering techniques allow us to implement a versioned XML data store with minimal efforts by employing a technology stack that is available so far, namely, version control systems and XML databases. Theoretical models and engineering techniques are supported by the robust implementation: xSVN introduced in Section 2.5 which is the core of the TNTBase system.

We will start with trivial definitions followed by the classical versioning model and concluding the theoretical part with the notion of an XML-enabled repository. Further we map the theoretical concept to the engineering field (Section 2.4) by describing the key engineering challenges and simplifications that can be undertaken to reuse existing technologies and diminish efforts on re-building components already well studied and available in the open source world. Finally, Section 2.5 will introduce the core part of the TNTBase system and describe the concrete engineering challenges encountered on the way towards building an XML-enabled repository. Section 2.6 concludes the chapter.

2.2 Fundamentals

2.2.1 Basic Concepts

Definition 2.1 (Components of Data Stores) Before going into defining concepts let us fix notations for common building blocks.

(1) A set of bytes \( \mathcal{B} \) is a set of all strings of length 8 over the alphabet \( \{0, 1\} \).

(2) \( \mathcal{I} \) is a set of identifiers.

(3) \( \mathcal{K} \) is a set of keys of a data store.

(4) \( \mathcal{M} \) is a set of metadata keys.

(5) \( \mathcal{N} \) is a set of names.

Definition 2.2 (A Data Store) We call a pair \( \mathcal{D} = (\mathcal{K}, \mathcal{C}) \) a data store, if \( \mathcal{C} \) is a function \( \mathcal{C} : \mathcal{K} \rightarrow \mathcal{B}^* \) called a data store function. We call \( \mathcal{K} \) a set of admissible (by the data store \( \mathcal{D} \)) keys. Additionally, an image of \( \mathcal{C} \) we call documents of the data store \( \mathcal{D} \).

\( \mathcal{D} \) is the set of all data stores.

Example 2.1 (A File System) For example, traditional disk file systems are data stores \( (\mathcal{K}, \mathcal{C}) \), where \( \mathcal{K} \) is a set of all existing file system paths and \( \mathcal{C} \) maps a file system path to the contents of a directory or a file.

Example 2.2 (An SQL Database) Every table in an SQL database has a primary key. Thus, SQL databases are data stores \( (\mathcal{K}, \mathcal{C}) \) where tuples consisting of a database name, a table name and a primary key are the set \( \mathcal{K} \), and \( \mathcal{C} \) maps a key \( k \) to a corresponding row in the table.
2.2 Fundamentals

The following concept of FS-trees share some similarities with the concept of the same name defined in Normen Müller’s Ph.D thesis [Müll10b]. However, the latter was developed for version control system (VCS) working copies, whereas the FS-concept described in this section concerns the VCS repository-side data store formalism. Overall two sides of the formalism constitute a full rigorous VCS theory.

**Definition 2.3 (An FS-tree)** An FS-tree is a tuple $T = \langle D, F, E, v^r, n, c \rangle$, where

1. $T = \langle V, E, v^r \rangle$ is a tree with root $v^r \in D$, where $V = D \cup F$ is a set of vertices, where $D \cap F = \emptyset$, and $E$ is a set of directed edges. Elements of $D$ are called directory nodes of $T$, and elements of $F$ are called file nodes of $T$.

2. If $\langle v, w \rangle \in E$ then $v \not\in F$, in other words $\forall v \in F: \text{outdeg}(v) = 0$, that is all $v \in F$ are tree leaves.

3. $n$ is a labeling name function: $n : E \rightarrow \mathbb{N}$ with a property that for all $\langle v, w_1 \rangle, \langle v, w_2 \rangle \in E$ we have $n(\langle v, w_1 \rangle) \neq n(\langle v, w_2 \rangle)$ if $w_1 \neq w_2$.

4. $c$ is a content function: $c : F \rightarrow \mathbb{B}^*$

We denote the set of all FS-trees by $\mathbb{T}$.

**Lemma 2.1** Every file system tree without hard or symbolic links can be represented as an FS-tree.

**Proof:** Let us build an FS-tree $T$ for a file system tree $t$. Let $D$ be a set of directories in $t$, and $F$ is a set of files in $t$. $v^r \in D$ is the root directory of $t$. For each directory $d$ in $t$ and its child $c$ we have an edge $\langle d^r, c^r \rangle$ in $E$, and a $n(\langle d^r, c^r \rangle)$ is the name of $c$. There are no edges that start with node $v \in F$, thus satisfying item 2.3.2 (files cannot have children). The content function is the function that maps a file $f_T$ to the contents of the corresponding file $f$.

**Definition 2.4 (An FS-tree Path)** Let $T = \langle D, F, E, v^r, n, c \rangle$ be an FS-tree. An FS-tree path is a function $P : V \rightarrow \mathbb{N}^*$ defined for every node $v \in V$ so if $\langle v^r, v_1, v_2, ..., v_n = v \rangle$ is a path from the root $v^r$ to the node $v$ then $\mathbb{P}(v) = \langle n(\langle v_1, v_2 \rangle), ..., n(\langle v_{n-1}, v_n \rangle) \rangle$. We denote the set of all paths by $\mathbb{P}$.

**Lemma 2.2** An FS-tree path uniquely defines a node of the FS-tree.

**Proof:** This is a direct corollary from Definition 2.3.3.

**Definition 2.5 (An FS-enabled Data Store)** We say that a data store $D = \langle \mathbb{N}, c \rangle$ represents an FS-tree $T = \langle D, F, E, v^r, n, c \rangle$ if and only if there is an injective function $\phi : F \rightarrow \mathbb{K}$ such that $c(v) = c(\phi(v))$ for all $v \in F$. Such a function $\phi$ is called an fs-mapping function of $T$. A data store $D = \langle \mathbb{K}, c, T, \phi \rangle$ is called an FS-enabled data store.
2. AN XML-ENABLED REPOSITORY

Example 2.3 (Subversion Repository) Let us consider a file system tree $t$ with a root directory $D$ that is committed to an SVN repository with the base URI $u$. Then the SVN repository at any given point in time can be represented as a data store $D = \langle \mathcal{K}, \mathcal{C} \rangle$, where $\mathcal{C}$ corresponds to the svn cat command that takes an argument in the form of $u/\pi$ forming the set $\mathcal{K}$, where $\pi$ is one of the file paths within a repository, and $u$ is a repository url.

We can say that a data store represented by an SVN repository that contains a directory $D$ represents an FS-tree induced (see Lemma 2.1) by the file system tree of the directory $D$. To prove that we have to construct an \textit{fs-mapping function}.

For all $v \in F$ there is a path $\langle v_1 = v^r, v_2, ..., v_n = v \rangle$ from the root node $v^r$ to $v$ (by definition of a tree). So we can define the function $\phi$ as $\phi(v) = u + / + n((v_1, v_2)) + / + ... + / + n((v_{n-1}, v_n))$, where "/" is a path separator. For example, a file \texttt{sigmod2011.tex} which has a directory \texttt{papers} as its parent and committed to repository \url{http://repos.tntbase.org} will be accessible by calling \texttt{svn cat http://repos.tntbase.org/papers/sigmod2011.tex}.

2.2.2 VCS Concepts

We can consider Example 2.3 as a guide to further concepts. However, in that example we looked at the SVN repository from a user perspective. Now we are going to show how the data store model is represented internally. Note that even though VCS formalism and naming of data structures has been inspired by SVN's Berkeley DB-based backend data store\footnote{http://svn.collab.net/repos/svn/trunk/subversion/libsvn_fs_base/notes/structure}, the model of a VCS presented here is much more general.

One of the central space-efficiency measures of VCS is the ability to store past revisions via their differences to the newest ones. So we need to model the basic operations of differencing and patching. Since we do not want to commit to a particular implementation strategy (see Appendix B.7 for elaboration), being it a character-based, line-based or XML differencing algorithm, we just generally define a pair of \textit{diff/patch} functions.

**Definition 2.6 (Diff/Patch Functions)** Any pair of functions $(\delta, \pi)$ are called \textit{diff/patch} functions on the data format $D \in \mathcal{B}^*$ if and only if\footnote{Here we refer to the intuitive understanding what a data format is. For example, it can be binary, text or XML}

1. $\pi(y, \delta(x, y)) = x$ for any $x, y$ from $D$.
2. $\delta(\pi(y, \Delta), y) = \Delta$ for any $y$ from $D$ and $\Delta$ from $\mathcal{B}^*$.

**Example 2.4 (Unix Diff)** For the text data format there exist Unix \textit{diff/patch} command line utilities. For example if \texttt{1.txt} contains text \texttt{Line1} and \texttt{2.txt} contains \texttt{Line 2} then the result of \texttt{diff 1.txt 2.txt} will be:

```
1a2
> Line 2
```
2.2 Fundamentals

This is a line-based diff algorithm. Applying the patch utility to 2.txt patch -R 2.txt diff.txt will result in contents of 2.txt being Line 1, thus satisfying the clause 1 of Definition 2.6. The clause 2 can be verified analogously.

For further formalism we will fix a pair of diff/patch functions \((\delta, \pi)\). With these we can model a repository as a sequence of revision trees:

**Definition 2.7 (A Revision Tree)** A revision tree is a tuple \(\mathcal{R} = \langle T, \eta, \tau, \rho \rangle\) where

1. \(T = \langle D, F, E, v^r, n, c \rangle\) is an FS-tree.
2. \(\eta : V \rightarrow I\) is an injective function that we call a node identification function, and its image is a set of revision tree node identifiers.
3. \(\rho\) is a natural number that we call a revision number. This number induces a trivial revision function \(\rho : V \rightarrow \mathbb{N}\) such that for all \(v \in V : \rho(v) = \rho\).
4. \(\eta^\rho : V \rightarrow I \times \mathbb{N} \times \mathbb{Z}\) is the function \(\eta^\rho(v) = \langle \eta(v), \rho, \chi \rangle\) for all \(v \in V\) where \(\chi \leq \rho\) is some natural number called copy identifier of node \(v\). \(\eta^\rho\) is called a node revision identification function, and its values are called node revision identifiers.
5. \(\tau : F \rightarrow \Sigma\) is a node revision type function where \(\Sigma = \{\text{fulltext}, \delta\}\) is a fixed set of types.

The intuition behind the \(\delta\) node revision type is that the corresponding node revision represents a difference (\(\delta\)) between the current node revision and some full contents (\(\text{fulltext}\)) of another node revision. \(\delta\) is a result of applying a diff function on two \(\text{fulltexts}\). On the other hand, a \(\text{fulltext}\) of a \(\delta\)-typed node revision can be obtained by applying a patch function on the current \(\delta\) and a \(\text{fulltext}\) the current delta has been produced from.

Since we will operate on node revision identifiers quite extensively we introduce a symbol for the node revision identifiers domain: \(\mathbb{I} = I \times \mathbb{N} \times \mathbb{Z}\).

**Example 2.5 (A Revision Tree as a VCS Revision)** Going a bit ahead, we can consider the following example that will be a guide to the next definition. A revision tree can be considered as a VCS repository tree for a particular revision \(\rho\), like a snapshot of a repository at some point in time, whereas each node of a tree can be uniquely identified by some internal value of function \(\eta^\rho\) across the whole repository. File contents are represented by a function \(c\) of the corresponding FS-tree and the contents can either be a \(\delta\) against other revisions or a \(\text{fulltext}\) representing the content of a file in question – the function \(\tau\) defines whether content is a \(\delta\) or a \(\text{fulltext}\).

Now we are ready to define one of the crucial concepts for this thesis: a repository.
2. AN XML-ENABLED REPOSITORY

Definition 2.8 (A Repository) A repository is a directed graph \( \mathcal{R} = (\{ \mathcal{R}_0, ..., \mathcal{R}_n \}, \mathcal{E}, s, \psi) \) where:

1. \( \mathcal{R}_i = (\mathcal{I}_i, \eta_i, \tau_i, \rho_i) \) with \( \rho_i = i \) and \( \mathcal{I}_i = (D_i, F_i, E_i, v_i^r, n_i, c_i) \) is an FS-tree. Additionally, \( D_i \cap D_j = \emptyset, F_i \cap F_j = \emptyset \) and \( E_i \cap E_j = \emptyset \) for all \( i \neq j \).

2. The set \( V_\mathcal{R} \) of vertices is \( D_\mathcal{R} \uplus F_\mathcal{R} \), where \( D_\mathcal{R} = \bigcup_{i=0}^{n} D_i \) and \( F_\mathcal{R} = \bigcup_{i=0}^{n} F_i \), thus \( V_\mathcal{R} = \bigcup_{i=0}^{n} V_i \) and \( D_\mathcal{R} \cap F_\mathcal{R} = \emptyset \).

3. The set of edges \( E_\mathcal{R} \) is \( \bigcup_{i=0}^{n} E_i \uplus \mathcal{E} \), where \( \mathcal{E} \) is an additional set of edges that start with vertices from \( D_\mathcal{R} \) such that for all \( (v_1, v_2) \in \mathcal{E} \) where \( v_1 \in D_\mathcal{R}, v_2 \in V_\mathcal{R} \) holds \( \rho(v_1) > \rho(v_2) \). Less formally it connects vertices from different revision trees, from newest to oldest. We call edges \( \mathcal{E} \) cross-revision edges of the repository \( \mathcal{R} \).

4. \( \eta_\mathcal{R} : V \to \mathbb{I} \) is an induced node identification function that is defined as: \( \eta_\mathcal{R}(v_i) = \eta_i(v_i) \) where \( v_i \in V_i \).

5. \( \eta_\mathcal{R}^r : V_\mathcal{R} \to \mathbb{I} \) is an induced node revision identification function that is defined as: \( \eta_\mathcal{R}^r(v_i) = \eta_i^r(v_i) \) where \( v_i \in V_i \).

6. \( n_\mathcal{R} : E_\mathcal{R} \to \mathbb{N} \) is an induced labeling name function with a property that for all \( (v, w_1), (v, w_2) \in E_\mathcal{R} \) we have \( n_\mathcal{R}((v, w_1)) \neq n_\mathcal{R}((v, w_2)) \) if \( w_1 \neq w_2 \).

7. \( c_\mathcal{R} : F_\mathcal{R} \to \mathbb{B}^* \) is an induced content function such that \( c_\mathcal{R}(v) = c_i(v) \) for all \( v_i \in F_i \) where \( i \in \{1..n\} \).

8. For all \( f \in F_\mathcal{R} \) holds \( \tau(v) = \text{fulltext} \).

9. \( \psi : \mathcal{E} \to \mathbb{N} \) is a copy identification function defined on the cross-revision edges. When we omit its value (including examples) we define that it has its default value \(-1\).

10. \( s : F_\mathcal{R} \to \mathbb{I} \) is a node successor function defined on those \( f \in F_\mathcal{R} \) that have \( \tau(f) = \text{delta} \). This function defines a successor node revision identifier of a delta file nodes within a repository. If \( s(f) = (i, r, x) \), then \( \rho(f) < \rho_x \). Informally, a successor function points to nodes in the successive revision trees.

\( \mathcal{R}_\mathcal{R} = \mathcal{R}_n \) is called a HEAD revision tree of a repository \( \mathcal{R} \). Additionally, we denote the set of all repositories by \( \mathcal{R} \).

For brevity, we call revision trees as well as revision numbers revisions. It is usually clear from context what concept is precisely meant. Otherwise, the exact meaning will be mentioned explicitly. Vertices from \( \mathcal{R} \) are called node revisions. Additionally, when mentioning node revision identifiers we will write them as \( (i, r, x) \) or just \( (i, r) \) when the copy identifier has its default value \(-1\). In all the other cases the value is non-negative and we will explicitly mention it.

Lemma 2.3 Every repository is an acyclic graph.
2.2 Fundamentals

Proof: This property directly follows from 2.8.3 because the $i$-trees are trees and additional edges $E$ connect only vertices between revision trees in one direction. □

Lemma 2.3 allows us to represent every repository as an acyclic directed graph (a DAG for further references).

Lemma 2.4 For all $v_1, v_2 \in V_R$ where $v_1 \neq v_2$ holds $\eta_R^v(v_1) \neq \eta_R^v(v_2)$, i.e. a revision identification function uniquely identifies every vertex $v$ from $V_R$.

Proof: If $v_1$ and $v_2$ belong to the same set $V_i$ then due to injectivity of $\eta_i$ the first parts of node revisions identifiers will be different. If $v_1$ and $v_2$ belong to different revision trees then the second part of node revision identifiers will be different since two different revision trees have different revision numbers. □

Lemma 2.4 allows us to give the following definition:

Definition 2.9 (Node revision lookup) Node revision lookup is an injective function $\ell: R \times I \rightarrow V_R$ that for a given repository $R$ and a node revision identifier returns a node from $R$. Instead of $\ell(R, (i, \rho, \chi))$ we will sometimes write $\ell R / i @ \rho \chi$ or just $\ell / i @ \rho \chi$ if it is clear what repository we are talking about.

Definition 2.10 (Fulltext content) Fulltext content of a file node $v$ in repository $R$ is a function $c_f: F_R \rightarrow B^*$ which is defined recursively:

$$c_f(v) = \begin{cases} c(v), & \text{if } \tau(v) \text{ is fulltext} \\ \pi(c_f(\ell(R, s(v))), c(v)), & \text{if } \tau(v) \text{ is delta} \end{cases}$$

Essentially, the fulltext content function obtains contents of a specific file node by applying deltas using a patch function against fulltext content of a successive file node that a successor function points to. The file node that is pointed to can be a delta too, therefore a recursive application of a fulltext content function is performed again. We do it until we encounter a fulltext node, and the next lemma will show that this will always happen.

Lemma 2.5 Function $c_f$ (see Definition 2.10) is correctly defined, that is the recursive application of $c_f$ will terminate.

Proof: Assume we have a node $f_i \in F_i \subset F_R$ for which $\tau(f) = \text{delta}$, then by clause 10 of Definition 2.8 we have a successor function $s$ that points to some file node $f_s$ in one of the successive revisions, which in order can also point to another successive tree if $\tau(f_s) = \text{delta}$, and so on until we encounter a node with the fulltext type. Since the number of revisions is finite, a successor function if always defined on delta nodes and $R_R$ contains only fulltext file nodes according to 2.8.3 we can conclude that the number of recursive applications of $c_f$ is also finite (it does not exceed $\rho_R - \rho_i$, where $\rho_R$ is a revision number of the HEAD revision tree). □
Example 2.6 (A Repository) Let us look at Figure 2.1 that represents a repository \( R \). It contains three consecutive revisions, that is three revision trees that are interlinked in one direction with red edges from the set \( E \). The intuition behind a repository is that any modification to an FS-tree (which is a subpart of any revision tree, see Definition 2.7) will imply only changing those parts of a revision tree that are affected by the change, that is the node itself and its ancestors. Unmodified nodes will remain as they are, without a necessity to be copied over, thus saving space and time for an operation. Modified nodes are marked with circles with a blue border, whereas green ones denote those nodes that were affected by those changes. A filled blue node is a node added to the revision 2. Letters represent node identifiers (see definition 2.7) that are common across revisions. However, node revisions identifiers are different for all nodes across the repository. For example, the node \( a \) have node revision identifiers \( \langle a, 0 \rangle \), \( \langle a, 1 \rangle \), and \( \langle a, 2 \rangle \) for revisions 0, 1 and 2, correspondingly. On the other hand, nodes \( a \) and \( b \) in revision 1 also have different revision identifiers: \( \langle a, 1 \rangle \) and \( \langle b, 1 \rangle \). A red edge from revision 1 to 0 has the semantics that a file node \( d \) has not been modified in revision 1 and still belongs to the directory with node identifier \( c \), hence the red edge \( \langle c, 1 \rangle \rightarrow \langle d, 0 \rangle \). The same is for \( \langle a, 2 \rangle \rightarrow \langle b, 1 \rangle \) and \( \langle a, 2 \rangle \rightarrow \langle c, 1 \rangle \). As an example of node revision lookup we can say that \( \ell/f@2 \) will look up a newly added node \( f \) in revision 2.

Finally, let us look at the orange labels of edges. They represent a labeling name function of repository \( R \). It is intuitively easy to see that an initial revision contained a file \texttt{readme.txt} and a directory \texttt{papers}. The latter contain two files: \texttt{MKM.tex} and \texttt{VLDB.tex}. The HEAD revision contains \texttt{readme.txt} and \texttt{VLDB.tex} changed in revision 1 leaving \texttt{MKM.tex} untouched. Additionally, a new file \texttt{index.html} has been added in revision 2.

Definition 2.11 (Revision checkout) Revision checkout is a function \( \emptyset : R \times N \rightarrow T \) that constructs an FS-tree from a repository \( R \) for a given revision number \( \rho \) with an additional node labeling function \( \zeta : V \rightarrow \mathbb{I} \). We will denote a resulting FS-tree by \( T_\rho \) which we will call a checked out FS-tree. Revision checkout is defined
2.2 Fundamentals

Figure 2.2: A HEAD revision checkout

recursively as follows:

(1) We take a DAG with all nodes and edges reachable from \( v^r_\rho \) - a root of \( \mathcal{R}_\rho \). We denote this graph as \( \mathcal{D}_{v^r_\rho} \).

(2) From \( \mathcal{D}_{v^r_\rho} \) we build an FS-tree as follows:

(i) We add \( v_r \) to vertices plus all adjacent to \( v_r \) nodes \( w_i \) where \( i \in \{0..K\} \) for some \( K \) where \( K \) is a number of adjacent vertices. We add edges that connect \( v_r \) with \( w_i \) to a resulting FS-tree preserving labeling information and a type (whether it is a directory or a file).

(ii) For all \( w_i \in F_T \) we define a content function \( c_T(w_i) = c_f(w_i) \).

(iii) For all \( w_i \in D_T \) we recursively add edges by applying these steps to DAGs \( \mathcal{D}_{w_i} \).

(iv) For all \( v_T \in V_T \) we define a labeling function \( \zeta(v_T) = \langle \eta_R(v), \rho(v), \chi \rangle \) where \( v \) is a corresponding vertex from \( V_R \) and \( \chi \) is a copy identifier obtained in the following way. If \( \langle e_1, ..., e_n \rangle \) is a path from \( v^r_\rho \) to \( v \) where \( e_i \in E_R \), then if there is \( \epsilon = e_i \in E \) such that \( \psi(\epsilon) \neq -1 \) and there is no \( e_k \in E, k > i \) such that \( \psi(e_k) \neq -1 \), then \( \chi = \psi(\epsilon) \). Otherwise, \( \chi = -1 \).

We have defined directory and file nodes, edges with the labeling function and a content function which is enough to completely represent an FS-tree.

Given a definition of a revision checkout it is straightforward to introduce a definition of a path of a repository node.

**Definition 2.12 (A Repository Path)** Let \( v_R \) be a node of the repository \( \mathcal{R} \) and \( \rho \) be some revision number. **Repository path** is a function \( P : \mathbb{R} \times V_R \rightarrow \mathbb{R} \) that is defined as follows. If exists \( v \in \mathcal{P}_\rho \) such that \( \ell(\mathcal{R}, \zeta(v)) \) is defined, then the **repository path** of \( v_R \) in revision \( \rho \) is \( P(v) \), otherwise we say that the node \( v_R \) does not exist in revision \( \rho \).
Example 2.7 (HEAD revision checkout) To fortify our understanding of Definition 2.11 let us consider Figure 2.2 which represents a checkout of the HEAD revision of the repository \( \mathcal{R} \). In the first step we select a DAG \( D_{v1} \) for revision 1 with the root node that has \( a \) as a node identifier. In the second step we “unwrap” DAG to represent an FS-tree. The fact that node revisions \( \langle a, 1 \rangle \), \( \langle c, 1 \rangle \) point to the same node revision \( \langle b, 0 \rangle \) means that in revision 1 the /backup/papers directory became a copy of the /papers directory which is represented by the value 0 of the copy identification function for second edge with the name papers. In revision 1 these two folders share the same content, but in further revisions their contents may start diverging.

Values of the labeling function \( l \) are shown in red. Note that some values are repeated not taking into account the copy identifier (e.g. \( \langle d, 0, -1 \rangle \) and \( \langle d, 0, 0 \rangle \)) meaning that directory/file nodes contents is the same – one is just a copy of another. Additionally, those nodes that have their labels with the second component equal to 1 represent that their counterparts in the repository have been modified in revision 1 (the green nodes on Figure 2.2).

Example 2.8 (Different data store models) In Figure 2.3 we can see two data store models: the model presented above in (1) and a naive data store model without cross-revision edges at (2). We can see that in the first case we use only 10 nodes for representing 3 revisions whereas in the second case we use 16 nodes. Likewise we use 11 and 13 edges, correspondingly. Even for such a simple example we can see savings in storage. For repositories with bigger trees and larger number of revisions those savings become even more apparent.

As we informally saw in examples 2.6 and 2.7 a repository model can represent trees
and modifications on them. However, we have not yet dealt with a problem of how a set of revision trees is expanded over time, nor have we considered any properties of repository modifications and which FS-trees are represented by revision trees, a repository itself and revision checkouts. All of these are the subject of the next section.

2.2.3 Operations on a Repository

In the previous section we have formally defined a concept of a repository and got some intuition how this model can be utilized for representing a file system tree that changes over time. So far we have not dealt much with node revision types from $\mathcal{X}$. Operations on a repository that we are about to introduce will make use of them. First of all, we will work out formalism for representing FS-tree modifications, then repository modifications. Afterwards we will show relations between those two kinds of modifications.

![Operations on FS-trees](image)

**Figure 2.4:** Operations on FS-trees

We can distinguish the following types of operations on FS-trees:

1. Adding an FS-tree or a file node to a directory
2. AN XML-ENABLED REPOSITORY

(2) Deleting a subtree or a file node

(3) Copying a subtree or a file node to another directory

(4) Modifying contents of a file node

The tree modification operations are explained on Figure 2.4. The figure does not contain a file node contents modification since that does not require reorganizing of the tree, but only redefining the content function for the file node in question. The moving operation can be represented as copying to the target and deleting from the source. More efficiently it can be done by rearranging edges – that is how moving on file systems is performed. But for simplicity we will not focus on it.

Let us define names and signatures of operations:

(1) \( \text{add}_T : T \times P \times N \times T \rightarrow T \) given an FS-tree, a path of its directory node, an edge name and an FS-tree to be added produces a resulting FS-tree.

(2) \( \text{del}_T : T \times P \rightarrow T \) given an FS-tree and a path of its node produces a resulting FS-tree without a subtree with the root in the given node.

(3) \( \text{copy}_T : T \times P \times N \times P \rightarrow T \) given an FS-tree, a path of the node to be copied, a name for a new edge and a path of the destination directory node produces a resulting FS-tree.

(4) \( \text{mod}_T : T \times P \times \mathcal{B}^* \rightarrow T \) given an FS-tree, a path of the file node and its new content, modifies a content function of an initial FS-tree so that if \( T' = \text{mod}_T(T, f, b) \) then \( c'(f) = b \) and \( c'(f') = c(f') \) for all \( f \neq f' \) in \( T \) where \( c' \) is a content function of \( T' \).

Operations on FS-trees are quite simple and intuitively easy to understand when keeping in mind that every file system tree can be represented as an FS-tree (recall Lemma 2.1). Operations on repositories are more complex. Let us define them and fortify our understanding by illustrations. It is important to realize that the intention behind a repository is that it must always preserve snapshots of data for any given revision number, thus modifications to a repository should lead to a creation of a new repository tree that tries to diminish data duplication.

Definition 2.13 (A Repository Operation) A repository operation on a repository \( \mathcal{R} \) with \( n \) revisions is a function in the form \( \text{op}_{\mathcal{R}} : \mathcal{R} \times \ldots \rightarrow \mathcal{R} \) that produces a repository \( \mathcal{R}' \) with the following properties:

(1) \( \mathcal{R}' \) contains a new revision tree \( \mathcal{R}_n \), where \( \mathcal{R}_n = \langle T_n, \eta_n, \tau_n, \rho_n \rangle \) with \( \rho_n = n \) and \( T_n = (D_n, F_n, E_n, v^r_n, n_n, c_n) \) is an FS-tree.

(2) The diff/patch functions remain unchanged.

(3) \( V_{\mathcal{R}} \subset V_{\mathcal{R}'} \) and \( E_{\mathcal{R}} \subset E_{\mathcal{R}'} \).
(4) For all $v \in F_n$ for which there exists $w \in F_R$ such that $s'(w) = v$ holds $\tau_R(w) = \text{delta}$. 

Informally speaking, a repository operation acts on a particular repository with additional parameters and produces a new repository with an added repository tree. Additionally, existing vertices and edges are preserved and augmented with the new ones. Moreover, a node revision type function is modified in such a way so that every file node from previous revisions that points via a successor function to a file node in the $n$-th revision has to have a $\text{delta}$ node revision type.

In the current model we define node revision types only for file nodes. However, it is also make sense to define them for directories too because large directories have to hold a long list of their children. Having directories $\text{deltified}$ would also give us some savings in storage space. But taking into account that it is not that important for our formalism we intentionally do not focus on directory deltification mechanisms in order to keep the model relatively simple.

Now let us define concrete repository operations that can be encountered in many version control systems.

### 2.2.3.1 Adding an FS-tree to a Repository

**Definition 2.14 (Addition to a repository)** Addition to a Repository $\mathcal{R}$ with $n$ revisions to a directory node with path $p \in \mathcal{P}$ as an entry with name $N \in \mathcal{N}$ is a repository operation $\text{add}_R : \mathcal{R} \times \mathcal{P} \times \mathcal{N} \times \mathcal{T} \to \mathcal{R}$ that produces a repository $\mathcal{R'}$ with the following properties:

1. There exists node $v \in \mathcal{T}_n^\mathcal{R}$ such that $\mathcal{P}(v) = p$.
2. Revision trees $\{\mathcal{R}_0...\mathcal{R}_{n-1}\}$ remain unchanged.
3. A new revision tree $\mathcal{R}_n$ is created according to the following rules:

   (i) First we get a checked out FS-tree $\mathcal{T}_n^\mathcal{R}$ and find a node $y \in \mathcal{T}_n^\mathcal{R}$ such that $\mathcal{P}(y) = p$.

   (ii) Let $\langle v_1, ..., v_n = y \rangle$ be a path to the node $y$. Then $\langle \zeta(v_1) = \langle \eta_1, i_1, \chi_1 \rangle, ..., \zeta(v_n) = \langle \eta_n, i_n, \chi_n \rangle \rangle$ is a sequence of node revision identifiers.

   (iii) We add to $\mathcal{R}_n$ nodes $\langle v'_1, ..., v'_n \rangle$ corresponding to $\langle v_1, ..., v_n = y \rangle$ with node revision identifiers $\langle \eta_i, n, \chi_i \rangle$ adding to $E_n$ edges between $v'_i$ and $v'_{i+1}$ where $i \in \{1..n-1\}$. Also if there is an edge $\langle v_i, w \rangle$ where $w \neq v_{i+1}$ then we add a cross-revision edge $\varepsilon = \langle v'_i, \ell(\mathcal{R}, \zeta(w)) \rangle$ and $\psi(\varepsilon) = -1$.

   (iv) To $v'_n$ we “connect” an added FS-tree $\mathcal{T}_a = \langle D_a, F_a, E_a, v'^*_a, n_a, c_a \rangle$ so that all its nodes will get node identifiers that do not yet exist in $\mathcal{R}$ and default copy identifiers. An additional edge $\langle v'^*_n, v'^*_n \rangle$ with labeling $n(\langle v'^*_a, v'^*_a \rangle) = N$ will “attach” $\mathcal{T}_a$ to a node $v'_n$ within $\mathcal{R}_n$. Also for all $f_a \in F_a$ we define $\tau(f_a) = \text{fulltext}$. 

25
We have defined an additional revision tree, cross-revision edges and node identifiers for the newly added file nodes. That allows us to define a new repository $\mathcal{R}' = \text{add}_R(\mathcal{R}, p, N, \mathcal{T}_a)$.

**Figure 2.5: Adding to the repository**

Figure 2.5 provides a simple example of adding a directory with the name $N = \text{"abstracts"}$ to the root of the repository (that is why the second argument is an empty sequence). Note that since the directory $\text{papers}$ has not been affected, we created a cross-revision edge $\langle a, 1 \rangle \to \langle b, 0 \rangle$. Newly added file nodes $\langle f, 1 \rangle$ and $\langle g, 1 \rangle$ have a fulltext node revision type as denoted by the italic letters $f$ on the figure.

### 2.2.3.2 Deleting from a Repository

**Figure 2.6: Deleting from the repository**
Definition 2.15 (Deletion from a Repository) Deletion from a repository $\mathcal{R}$ with $n$ revisions of a node with path $p \in \mathcal{P}$ is a repository operation $\text{del}_R : \mathcal{R} \times \mathcal{P} \rightarrow \mathcal{R}$ that produces a repository $\mathcal{R}'$ with the following properties:

1. There exists a node $v \in T^n_\mathcal{R}$ such that $\mathcal{P}(v) = p$.
2. Revision trees $\{\mathcal{R}_0...\mathcal{R}_{n-1}\}$ remain unchanged.
3. A new revision tree $\mathcal{R}_n$ is created according to the following rules:
   - First we get a checked out FS-tree $T^n_\mathcal{R}$ and find a node $y \in T^n_\mathcal{R}$ such that $\mathcal{P}(y) = p$.
   - Let $\langle v_1,...,v_n = y \rangle$ be a path to the node $y$. Then $\langle \zeta(v_1) = \langle \eta_1, i_1, \chi_1 \rangle,...,\zeta(v_n) = \langle \eta_n, i_n, \chi_n \rangle \rangle$ is a sequence of node revision identifiers.
   - We add to $\mathcal{R}_n$ nodes $\langle v'_1,...,v'_{n-1} \rangle$ corresponding to $\langle v_1,...,v_{n-1} \rangle$ with node revision identifiers $\langle \eta_n, n, \chi_i \rangle$ adding to $E_n$ edges between $v'_i$ and $v'_{i+1}$ where $i \in \{1..n-1\}$. Also if there is an edge $\langle v_i, w \rangle$ where $w \neq v_{i+1}$ then we add a cross-revision edge $\varepsilon = \langle v'_i, \ell(\mathcal{R}, \zeta(w)) \rangle$ and define $\psi(\varepsilon) = -1$. Note that we do not add the node $y = v_n$ to $V_n$.

We have defined an additional revision tree and cross-revision edges. Since we do not add new nodes we do not have to create extra node identifiers. Thus the new repository $\mathcal{R}' = \text{del}_\mathcal{R}(\mathcal{R}, p)$ is defined.

Figure 2.6 provides an example of a file deletion. The file 1.tex is deleted from the directory abstracts. Since it does not affect the file 2.tex as well as the folder papers we have the cross-revision edges in revision 1 that point to the nodes in revision 0 that are unchanged in revision 1. Only those directory nodes are presented in revision 1 that were the ancestors of the file 1.tex. Thus a repository deletion operation does not actually remove content from a repository, instead it just removes a link to the deleted item in the new revision.

2.2.3.3 Copying within a Repository

Copying within a repository can, of course, be represented via addition of the sub-tree in question. However, there is a more efficient way to do that:

Definition 2.16 (Copying within a Repository) Copying within a repository $\mathcal{R}$ with $n$ revisions of a node with path $s \in \mathcal{P}$ to a directory node with path $t \in \mathcal{P}$ as an entry with name $N$ is a repository operation $\text{copy}_\mathcal{R} : \mathcal{R} \times \mathcal{P} \times \mathcal{P} \times N \rightarrow \mathcal{R}$ that produces a repository $\mathcal{R}'$ with the following properties:

1. There exist nodes $v, w \in T^n_\mathcal{R}$ such that $\mathcal{P}(v) = s$ and $\mathcal{P}(w) = t$.
2. Revision trees $\{\mathcal{R}_0...\mathcal{R}_{n-1}\}$ remain unchanged.
3. A new revision tree $\mathcal{R}_n$ is created according to the following rules:
2. AN XML-ENABLED REPOSITORY

(i) First we get a checked out FS-tree $T^n_R$ and find a node $y \in T^n_R$ such that $P(y) = t$.

(ii) Let $\langle v_1, ..., v_n = y \rangle$ be a path to the node $y$. Then $\langle \zeta(v_1) = \langle \eta_1, i_1, \chi_1 \rangle, ..., \zeta(v_n) = \langle \eta_n, i_n, \chi_n \rangle \rangle$ is a sequence of node revision identifiers.

(iii) We add to $R_n$ nodes $\langle v'_1, ..., v'_n \rangle$ corresponding to $\langle v_1, ..., v_n \rangle$ with node revision identifiers $\langle \eta_i, n, \chi_i \rangle$ adding to $E_n$ edges between $v'_i$ and $v'_{i+1}$ where $i \in \{1..n-1\}$. Also if there is an edge $\langle v_i, w \rangle$ where $w \neq v_{i+1}$ then we add a cross-revision edge $e = \langle v'_i, \ell(R, \zeta(w)) \rangle$ and $\psi(e) = -1$.

(iv) Additionally we create a cross-revision edge $e = \langle v'_n, \ell(R, \zeta(z)) \rangle$ with labeling $n(e) = N$ where $z \in V_R$ such that $P(z) = s$. Copy identification function is defined as $\psi(z) = \rho(z)$.

We have defined an additional revision tree and cross-revision edges, one of which represents a copied sub-tree with root at path $s$. The new repository $R' = \text{copy}_R(R, s, t, N)$ is defined.

![Figure 2.7: Copying in the repository](image)

We can see that in the way formalized in Definition 2.16 the copying operation becomes “cheap” since it does not imply copying the whole sub-tree but rather linking its root from the target folder. The concept of cheap copies exists now in many modern version control systems: for example, in SVN or Git. However, for instance, still popular CVS does not have this concept that makes copying an expensive operation. In fact, that is one of the major drawbacks of CVS.

Figure 2.7 exemplifies cheap copying where the papers folder is copied to backup under the different name pub. Note that the cross-revision edge with the name pub has copy identifier 0 since it represent the copy from revision 0. Another cross-revision edge
papers has its default copy identifier $-1$ since it does not represent a copy operation but just links to the unchanged directory. The new revision contains only two nodes: a target node and its parent. Storage space effectiveness is apparent.

Note that moving within a repository can be represented as copying of the source node to the target directory, and then deleting a source node.

### 2.2.3.4 Modifying File Contents in a Repository

The last basic repository operation that we are going to define is a modification of file contents in the space-efficient manner.

**Definition 2.17 (File Contents Modification)** File Contents Modification in the repository $R$ with $n$ revisions of a file node with path $p \in P$ and new content $\varsigma \in B^*$ is a repository operation $\text{mod}_R : R \times P \times B^* \rightarrow R$ that produces a repository $R'$ with the following properties:

1. There exists node $v \in T^n_R$ such that $P(v) = p$.
2. Revision trees $\{R_0, \ldots, R_{n-1}\}$ remain unchanged except the case defined below.
3. A new revision tree $R_n$ is created according to the following rules:
   (i) First we get a checked out FS-tree $T^n_{R_0}$ and find a node $y \in T^n_{R_0}$ such that $P(y) = p$.
   (ii) Let $\langle v_1, \ldots, v_n = y \rangle$ be a path to the node $y$. Then $\langle \zeta(v_1) = \langle \eta_1, i_1, \chi_1 \rangle, \ldots, \zeta(v_n) = \langle \eta_n, i_n, \chi_n \rangle \rangle$ is a sequence of node revision identifiers.
   (iii) We add to $R_n$ nodes $\langle v'_1, \ldots, v'_n \rangle$ corresponding to $\langle v_1, \ldots, v_n \rangle$ with node revision identifiers $\langle \eta_i, n, \chi_i \rangle$ adding to $E_n$ edges between $v'_i$ and $v'_{i+1}$ where $i \in \{1..n-1\}$. Also if there is an edge $\langle v_i, w \rangle$ where $w \neq v_{i+1}$ then we add a cross-revision edge $\varepsilon = \langle v'_i, \ell(R, \zeta(w)) \rangle$ and $\psi(\varepsilon) = -1$.
   (iv) We define a new content function $c_{R'}$ such that $c_{R'}(v'_n) = \varsigma$ and $c_{R'}(f) = c_{R}(f)$ for all $f \neq v'_n$.
   (v) $\tau_{R'}(v'_n) = \text{fulltext}$
   (vi) Additionally, if the copy identifiers of $v'_n$ and $y$ are equal, then:
   - We define $\varsigma_{R'}(y) = \delta(\varsigma, c_{R}(y))$
   - $\tau_{R'}(y) = \text{delta}$ and $\tau_{R'}(f) = \tau_{R}(f)$ for all $f \neq y$ and $f \neq v'_n$.
   - Since we changed a node revision type of $y$ from $\text{fulltext}$ to $\text{delta}$ we have to define a successor function (see Definition 2.8.10) for $y$ such that $s_{R'}(y) = \eta_{R'}(v'_n)$.

We have defined an additional revision tree and cross-revision edges. Also we have changed the content function of a repository and a node revision type of previous version of the modified file. After that we linked the deltified file node to its newest version so that the fulltext content of the old version can be obtained via the patch.
function as $\pi(c_{R'}(\ell(R', s(y))), c(y))$. Note that we deltify a previous revision of a node only when a modified node is just a successor of $y$, but not a copy of it. Finally, the new repository $R' = \text{mod}_R(R, p, \varsigma)$ is defined.

Figure 2.8: Modifying a file node in the repository

Figure 2.8 visually explains Definition 2.17. As in case with other defined repository operations only those nodes are presented in the new revision that are ancestors of the modified node and the modified node itself. Unchanged nodes are linked via cross-revision edges. What is new here is that a file node with the node identifier d now has a delta node revision type in revision 0 and its successor function points to its latest fulltext content.

Sometimes deltification of a node does not make sense. For instance, when the modified file is binary: nearly every diff function that is suitable for text-based deltification would produce a large delta which size is comparable to the file size itself. In such cases it makes more sense not to deltify the previous file revision but to leave its contents as fulltext. However, in our model we always perform deltification since it does not harm our theory and is a more general scenario.
2.2 Fundamentals

2.2.4 Relation between FS-trees and Repositories during a Repository Lifetime

Figure 2.9: Checkout and Modifications

Repository is a relatively difficult concept, unlike FS-trees which are intuitively understandable by any user who is familiar with the structure of file systems. However, there is a strong relation between those two concepts. In order to explain that we first have to introduce a concept of the FS-tree isomorphism.

Definition 2.18 (FS-tree Isomorphism) Two FS-trees \( T_1 = \langle D_1, F_1, E_1, v'_1, n_1, c_1 \rangle \) and \( T = \langle D_2, F_2, E_2, v'_2, n_2, c_2 \rangle \) are isomorphic if and only if:

1. There exists a graph isomorphism \( i \) such that two vertices \( v_1 \) and \( w_1 \) are adjacent in \( T_1 \) if and only if \( i(v_1) \) and \( i(w_1) \) are adjacent in \( T \).
2. AN XML-ENABLED REPOSITORY

(2) \( i(v'_1) = v'_2 \)

(3) \( d \in D_1 \) if and only if \( i(d) \in D_2 \).

(4) \( n_1(v) = n_2(i(v)) \) for all \( v \in V_1 \).

(5) \( c_1(v) = c_2(i(v)) \) for all \( v \in F_1 \).

We write \( \mathcal{T}_1 \simeq \mathcal{T}_2 \).

Example 2.9 (File System Folders Isomorphism) If one folder on the file system is a copy of another, then these two are isomorphic: the structure of folders is the same as well as the names of the descendant directories and files. Also file contents is the same. However, we cannot say that two folders are equal since e.g. in the POSIX file systems they are represented by different i-nodes. But for an end-user they look absolutely the same since he never sees the underlying data structure of file systems – that is what basically the isomorphism on FS-trees represents.

So far we have been concerned with operations on repositories. In other words we have considered transitions from one revision to the consecutive one. To understand how a repository starts its lifecycle consider a definition of the repository creation operation.

Definition 2.19 (A Repository Creation Operation) A Repository Creation Operation is a function \( C : \mathcal{T} \to \mathcal{R} \) that creates a repository \( \mathcal{R} = (\{R_0\}, \emptyset, s, \psi) \) from an FS-tree \( \mathcal{T} \), where

1. \( \mathcal{R}_0 = \langle \mathcal{T}', \eta, \tau, 0 \rangle \) is an initial revision tree, where
   - (i) \( \mathcal{T}' = \langle D', F', E', v', n', c' \rangle \) is an FS-tree that is isomorphic to \( \mathcal{T} \).
   - (ii) \( \eta \) is a node identification function that assigns unique node identifiers for all \( v \in V' \).
   - (iii) \( \tau(f) = \text{fulltext} \) for all \( f \in F' \).

2. Since all file node have the fulltext node revision type, the domain of the function \( s \) is an empty set.

3. Since \( \mathcal{E} \) is an empty set, the domain of \( \psi \) is an empty set as well.

Note that we can also consider creating a repository for a revision tree that contains just a single directory root node without any children. For example, that is how a fresh SVN repository looks like.

Definition 2.20 (Repository Representation) We say that a revision \( n \) of the repository \( \mathcal{R} \) represents the FS-tree \( \mathcal{T} \) if and only if \( \mathcal{L}(\mathcal{R}, n) \simeq \mathcal{T} \).

Now let us explain how repositories and FS-trees are related.
Lemma 2.6 Let \( T \) be an FS-tree, then \( \mathcal{C}(\mathcal{E}(T), 0) \simeq T \). Less formally, a checked-out FS-tree of the initial repository revision is isomorphic to the FS-tree that initiated the repository.

**Proof:** Since an initial revision does not contain any cross-revision edges, then the checked out FS-tree for revision 0 is an FS-tree of the revision tree 0 that, in turn, by Definition 2.19 is isomorphic to the FS-tree which initiated the repository. \( \square \)

**Definition 2.21 (Repository Commit)** Let \( \mathcal{R} = \langle \{ R_0, ..., R_n \}, E, s, \psi \rangle \) be a repository and \( T \) is an FS-tree such that \( \mathcal{C}(\mathcal{R}, n) \simeq T \). Then a **repository commit** is an operation on the repository \( \mathcal{R} \) after a modification of \( T \) under the following rules:

1. If \( T' = \text{add}_T(T, p, n, T_a) \), then \( \mathcal{R}' = \text{add}_{\mathcal{R}}(\mathcal{R}, p, n, T_a) \).
2. If \( T' = \text{del}_T(T, p) \), then \( \mathcal{R}' = \text{del}_{\mathcal{R}}(\mathcal{R}, p) \).
3. If \( T' = \text{copy}_T(T, s, n, t) \), then \( \mathcal{R}' = \text{copy}_{\mathcal{R}}(\mathcal{R}, s, n, t) \).
4. If \( T' = \text{mod}_T(T, p, \varsigma) \), then \( \mathcal{R}' = \text{mod}_{\mathcal{R}}(\mathcal{R}, p, \varsigma) \).

A repository commit is **admissible** if and only if the isomorphism condition in the beginning of the definition holds.

**Theorem 2.1** If we realized an admissible repository commit of \( \mathcal{R} \) with \( n \) revisions after the modification of the FS-tree \( T \), then a revision \( n + 1 \) of \( \mathcal{R}' \) represents \( T' \).

**Proof:** This is a corollary of the admissibility of the repository commit and the Definitions 2.14-2.17. \( \square \)

Figure 2.9 visualizes Theorem 2.1 for the commit from revision 0 to revision 1. We can see that the papers directory has been copied with the same name to the backup folder. If we look at consecutive FS-trees and their checked out FS-tree counterparts according to the FS-tree operations and the definition of the repository checkout we are able to see that they are indeed the “same” in out intuitive perception – they look the same, have the same set of nodes and edges along with the edges’ names. On the other hand they were obtained from different (although, related) sources. Therefore we cannot say that e.g. FS-trees \( T \) and \( T_0 \) are equal, but they are isomorphic.

**Corollary 2.1** Let \( T_0 \) be an FS-tree and \( \mathcal{R}_0 = \mathcal{C}(T_0) \) a repository initiated from \( T_0 \). Also \( \{ T_1, ..., T_n \} \) are \( n \) FS-trees obtained by consecutively applying modification operations starting from \( T_0 \) and \( \mathcal{R}' \) is a repository obtained by consecutive repository commits based on FS-tree modifications. That entails that revision \( i \) of \( \mathcal{R}' \) represents \( T_i \) for any \( i \in \{0..n\} \).

Corollary 2.1 gives us one of the main results of this Chapter: a single repository can represent multiple FS-trees without duplication of content unchanged through FS-tree modifications.
2. AN XML-ENABLED REPOSITORY

Note that we assumed that a repository commit implies just one trivial operation on the repository. However, in practice, a commit is typically a sequence of non-conflicting repository operations that are performed atomically. In other words, each commit is a repository transaction. We deliberately do not cover this topic since all the transactional functionality can be built on top of the data structures that we have presented so far: a new revision of the repository would be just a revision tree obtained by consecutive applying repository operations within the scope of one commit.

What we did not consider so far is that the content of repository FS-trees must be stored somewhere, but the content function of a repository does not specify that. Next section will bring us closer to real version control systems that utilize data stores for keeping information persistent.

2.2.5 Data Store Repository

Definition 2.22 (Data Store Repository) Let \( \mathcal{D} \in \mathbb{D} \) be a set of data stores, then a data store repository is a tuple \( \mathcal{D} = (\mathfrak{R}, \mathcal{D}, d, \kappa) \) where

1. \( \mathfrak{R} \) is a repository.
2. \( d : F_{\mathfrak{R}} \rightarrow \mathcal{D} \) is a data store function.
3. \( \kappa : F_{\mathfrak{R}} \rightarrow \mathcal{K} \) is a key-mapping function such that \( C(\kappa(f)) = c(f) \) for all \( f \in F_{\mathfrak{R}} \) where \( C \) is a content function of the data store \( D = d(f) \) from \( \mathcal{D} \).

\( \mathcal{D} \) is a set of data stores supported by a data store repository \( \mathcal{D} \).

The idea behind having multiple data stores is to use an appropriate data store for a particular data format. Although definition 2.22 is quite general, what we intend to have are two different data stores for XML content and other document types. The reason for it is that XML-databases in vast majority of use cases are better suited for storing XML since they provide XML-related functionality like validation, querying or indexing. However, the definition 2.22 is not bound to XML use case only. For instance, if we wanted to optimize our repository for storing JSON content, we could choose MongoDB \([\text{Mon}]\) as an additional data store.

Now we have worked out all concepts that are sufficient to continuing with elaborating on XML data stores and XML databases.

2.3 XML-enabled Repository (XeR)

2.3.1 XPath, XQuery Data Model

Before proceeding with the XML-related concepts we need to introduce the XPath, XQuery Data Model (XDM). XDM defined by W3C in \([\text{Ber+10}]\) is a data model for three W3C languages, namely XPath 2.0, XSLT 2.0 and XQuery 1.0. It serves as a definition for input and output of those languages that are closed with respect to XDM. In other words, XDM defines what information is accessible in the documents that are
2.3 XML-enabled Repository (XeR)

instances of XDM. XDM is based on XML Information Set specification, but was extended to meet requirements of XPath, XSLT and XQuery. An instance of XDM does not necessarily represent an XML document, but may, for example, represent a sequence of document nodes, empty sequences or just a single atomic value like an integer or a date.

2.3.2 Building an XML-enabled Repository

We denote by $X$ a set of all XDM instances.

**Definition 2.23 (XML Data Store)** We call XML data store $\mathcal{X} = \langle \mathcal{K}, \mathcal{C} \rangle$ that for all $c$ such that $\mathcal{C}(k) = c$ we have that $c \in X_D$, where $X_D$ is a set of all XDM document nodes.

Less formally, an XML data store maps a set of keys to XML documents. We denote all XML data stores by $\mathcal{X}$.

**Definition 2.24 (XML FS-subtree)** An XML FS-subtree $\mathcal{X}_\mathcal{T}$ of an FS-tree $\mathcal{T} = \langle D, F, v^r, n, c \rangle$ is a tuple $\mathcal{X} = \langle D, F, E, v^r, n, c \rangle$, where

1. $F, X$ is a subset of $F$, such that $c(F, X)$ are in $X_D$.
2. $E, X = E \setminus \{(v, w) : w \in X \setminus F_X\}$

**Lemma 2.7** Every XML FS-subtree is an FS-tree

**Proof:** Direct corollary of Definition 2.24

**Definition 2.25 (XML-enabled Repository)** An XML-enabled repository is a data store repository $\mathcal{X} = \langle \mathcal{R}, \mathcal{D}, d, \kappa \rangle$ where

1. $\mathcal{D}$ contains an XML data store $X_\mathcal{X}$.
2. $d(f) = X_\mathcal{X}$ if and only if $f \in F_X$, $c(f)$ is from $X_D$ and $\tau(f) = \text{fulltext}$.

Less formally, $\mathcal{X}$ supports multiple data stores among which one is an XML data store. And all fulltext XML file nodes are stored in that XML data store.

**Theorem 2.2** An XML data store $\mathcal{X}$ of the XML-enabled repository $\mathcal{X}$ with $n$ revisions represents an XML FS-subtree of $\mathcal{C}(\mathcal{X}, n)$. That is the XML data store of the XML-enabled repository contains the latest versions of all committed XML documents without losing their file system locations.

**Proof:** Every HEAD revision checked out FS-tree $\mathcal{T} = \mathcal{C}(\mathcal{X}, n)$ has a labeling function $\zeta : V_\mathcal{T} \rightarrow \mathcal{I}$. We have to construct a function $\phi : F_\mathcal{T} \rightarrow \mathcal{K}$ such that $c_{\mathcal{X}_\mathcal{T}}(v) = \mathcal{C}(\phi(v))$ for all $v \in F_\mathcal{T}$ where $\mathcal{X}_\mathcal{T}$ is an XML FS-subtree of $\mathcal{T}$ and $\mathcal{C}$ is a content function of $\mathcal{X}$. We know that every $v \in F_\mathcal{T}$ has the node in $\mathcal{X}$ that it corresponds to. This node is looked up by $\ell_\mathcal{X}(\zeta(v))$. Let us denote it by $w$. Its content is $c_\mathcal{X}(w)$ and by
2. AN XML-ENABLED REPOSITORY

Definition 2.22 of the data store repository $c_X(w) = \ell(\kappa(w))$. Since we consider the HEAD revision of $X$ it holds that $c_X(w) = c_Y(v)$ (fulltext content of $w$ is just equal to the value of the repository content function of $w$) and thus we found the function $\phi(v) = \kappa(w) = \kappa(\ell_X(\zeta(v)))$. □

2.4 Towards An XML-Enabled Repository

2.4.1 Methodology and General Architecture

Theorem 2.2 gives us theoretical foundations for designing and implementing an XeR which will be a basis for introducing a versioned XML-database in the next chapter. However, there are several reasons why an XeR is not yet that beneficial: file system information of XML documents is tightly coupled with XeR, that is there is no way to extract file system information from the XML data store directly. Secondly, an XML data store itself does not give us many benefits except guaranteeing well-formedness of XML documents it contains. Versioned XML databases introduced in the next chapter attack these issues. We will see that storing XML in the XML data store gives us huge potential of getting fine-granular and efficient access to XML data.

Efficient fine-grained access to XML documents is possible in theory (as shown in this and consecutive chapters), but in order to achieve it in practice with minimal amount of efforts we propose to employ an existing stack of technologies, namely version control systems and XML databases. This approach induces implementations that can be easily and transparently realized. The practical contribution gives an abstract, but at the same time comprehensive description of how it can be done with minimized amount of efforts. The approach can help avoiding common pitfalls and build your own XeR in the efficient manner on top of the technology stack of programmer’s choice that can be further extended to meet the developer/user needs. Of course, not every system is suitable for being a part of the approach. However, the class of such system is wide enough that we have much freedom for engineering decisions.

We have proven the feasibility of the approach by instantiating it to Subversion and Berkeley DB XML (Section 2.5) and in the next chapter we will show how this implementation can be used as a basis for further development towards a versioned XML database.

In general, as illustrated in Definition 2.22 a basic idea is to integrate an XML data store into the backend storage of a VCS and use it for storing XML documents. We shall call a product of integration of an XML data store $X$ into a VCS $V$ an XML-enabled repository of $V$ and $X$ or XeR$(V,X)$.

Figure 2.10 illustrates that an XML data store is embedded into an existing VCS $V$ and is used in conjunction with a default VCS storage (both data stores constitute a set of data stores $\emptyset$ from Definition 2.22) whereas a client remains unaware of the storage details.

Integration of an XML DB into a VCS can be smoother if both systems satisfy a number of properties (see Appendix B.2), however, it does not affect the generality of
2.4 Towards An XML-Enabled Repository

Now we are ready to discuss the XeR approach in detail. In order to build XeR we need to do the following (recall Definition 2.25):

- Pick such VCS $V$ that conforms to the repository model defined in Definition 2.8 and supports repository operations defined in Section 2.2.3.
- Provide a data store function $d$ that for all fulltext XML documents assigns an XML data store $X$, and a default data store of $V$ for other documents.
- Provide a key-mapping function $\kappa$ such that if $f$ is a fulltext XML file node, then $C(\kappa(f)) = c(f)$, where $C$ is a content function of $X$ and $c$ is a content function of $V$'s repository.\footnote{We assume that this is true for all repositories that are maintained by $V$.}

2.4.2 A Data Store Function

Realizing a data store function boils down to reliably identifying whether document is in the XML format or not, or distinguishing those XML documents that we are interested in. For example, storing XSD files in $X$ may not bring additional advantages since we are unlikely going to query them, although they are in the XML format. There are multiple approaches to defining for which files $d(f) = X$, and typically the best one is picked according to user requirements. Discussion on this topic can be found in Appendix B.1. For the rest of this chapter we assume that $d$ is defined and its definition has no influence on the overall approach.
2. AN XML-ENABLED REPOSITORY

Figure 2.11: A Commit Workflow

2.4.3 A Key-Mapping Function for Embedding $\mathcal{X}$ into $\mathcal{V}$

The only key element that is missing for integration $\mathcal{X}$ into $\mathcal{V}$ is a realization of a key-mapping function $k$ that maps repository node content to its physical location in the data store.

Figure 2.11 depicts a high-level abstraction of a commit process where multiple files are being committed to a repository. Our integration of $\mathcal{X}$ into $\mathcal{V}$ concerns only the $\mathcal{V}$’s storage layer, and therefore we do not focus on other layers of $\mathcal{V}$ that can be, for instance, a network or authentication layers. The rhomb on the figure denotes a data store function $d$. A key-mapping function $\kappa$ about which we will talk later will be responsible for defining a key for every XML file so that it can be stored and referenced in $\mathcal{X}$. If an XML file already existed in $\mathcal{X}$, a new version of it is stored in $\mathcal{X}$, whereas a previous revision is moved to a default storage of $\mathcal{V}$ by computing a reverse delta against the HEAD revision (recall the formal definition of the modification operation 2.2.3.4) – that is depicted by an arrow at the bottom of the figure.

A key-mapping function can be realized in multiple ways to satisfy Clause 3 of Definition 2.22. However, we are going to use our theoretical result of Lemma 2.4 that a revision identification function uniquely identifies every node from the repository. Let us fix a repository $\mathcal{R}$ in $\mathcal{V}$. Also because unmodified $\mathcal{V}$ is a data store repository (since every VCS has an underlying storage) we have an old key-mapping function $\kappa_0$. Now we are going to modify it only for the fulltext XML nodes in $\mathcal{R}$:

$$
\kappa(v) = \begin{cases} 
k(\eta_{\mathcal{R}}(v)) & \text{if } d(v) = \mathcal{X} \\
k_0(v) & \text{otherwise}
\end{cases}
$$
2.5 The XeR Implementation – xSVN

where \( k : I \rightarrow K \) is a function that maps node revision identifiers to data store keys. For example, if a node revision identifier is \( \langle i, 4, 0 \rangle \), then \( k \) can map it to a concatenated version separated by dashes: \( i-4-0 \). It is easy to see that \( k \) is an injective function due to injectivity of \( \eta^\rho_R \) (see Lemma 2.4).

We have shown how to implement a key-mapping function that guarantees absence of collisions due to its injectivity. By inheriting part of \( \kappa \) image from an initial version of \( V \), we can assure the same behavior for non-XML files and focus only on implementing storage functionality for \( X \), thus simplifying the overall implementation of an XML-enabled repository. Data consistency, durability, atomicity of a commit, etc. are those issues that an XeR must take care of. Ideally, it should be relatively easy to hook in existing mechanism of the initial VCS \( V \). Section 2.5 will cover a concrete realization of XeR by looking into more specific details and showing how practical issues have been solved.

2.5 The XeR Implementation – xSVN

xSVN is a Subversion on steroids. It is an implementation of the XML-enabled repository concept that is based on Subversion and Berkeley DB XML (DB XML), and is a core of a larger system – the versioned XML database TNTBase. Essentially xSVN is XeR \((V, X)\), where \( V = SVN \), \( X = DB XML \).

The following facts significantly simplified implementation efforts:

- SVN has a modular design for various backend data stores. Actually, unmodified version of SVN can make use of two different backend data stores – Berkeley DB and its own SVN FS. Such modular design is important for embedding yet another data store.

- Like a vast majority of version control systems, SVN has a notion of an atomic commit – either all or nothing is committed. So we inherit transactional support automatically.

- SVN also takes care of data integrity by preserving content checksums on a commit and verifying against them upon data retrieval – one thing less to worry about when building a robust system.

- DB XML is an embeddable database (designed to run in the same address space as the parent process) with transaction support.

- DB XML supports indexing as well as arbitrary metadata to be associated with XML documents which allowed us to perform a number of optimizations.

Before we continue describing the implementation challenges and solutions, we shall briefly summarize behavior of the implemented data store function. In other words, what files are considered by xSVN as XML.
2. AN XML-ENABLED REPOSITORY

2.5.1 What is an XML File for xSVN?

By default, all files that satisfy the pattern *.xml and *.omdoc are considered to be XML. The latter pattern is dictated by the fact that TNTBase is extensively used for documents in OMDoc format that have the omdoc extension. Also, by default, files with the svn:mime-type in set {text/xml, application/xml, application/omdoc+xml} are also considered to be XML. Generally file patterns and media types can be set up in the SVN’s server configuration area under section [global]. For example:

```plaintext
[global]
tnt.ignore=all=no
tnt.files=*.omdoc;*.xml;*.owx;
tnt.media-types=text/xml;text/xml+omdoc
```

`tnt.files` and `tnt:media-types` attributes override default ones. If one does not want to treat any files as XML he might want to set up the attribute `tnt.ignore-all` to `yes`. These parameters might be optionally set up only for a certain period of time affecting only those revisions that have been made in between.

2.5.2 Integration

The first design choice for xSVN is to use the Berkeley DB (BDB) [Bera] backend of SVN to integrate DB XML to. This choice has been made due to the observation that Berkeley DB and DB XML share the same environment and therefore can share transactions. This fact simplifies coordination of content operations and guarantees consistency and durability.

Note that both SVN and DB XML use BDB: SVN for storing repository data and DB XML for storing raw bytes of XML and indexes as well as for supporting consistency, recoverability, transactions, etc. As Subversion stores only HEAD revisions as fulltext and others as reverse deltas it perfectly conforms to the repository model we introduced earlier. Thus we just needed to implement a DB XML storage layer for fulltext XML nodes – all other repository data is retained in BDB (non-XML files, deltas, repository FS-tree information, properties, changes summary, etc.). Let us look at the situation in more detail.

The SVN BDB-based file system uses multiple tables to store various repository data like information about locks, revisions, transactions, files, directories, etc. (see Figure 2.12). In order to understand the integration process we need to look at the two BDB tables: the `representations` and `strings` tables used by SVN. The `strings` table stores only raw bytes of data and one entry of this table could be any of these:

1. file content or a delta that reconstructs file content from some other entry in the same table.

---

1Sharing of transactions means that a common transaction handler can be passed to API calls of both Berkeley DB and DB XML, and thus those API calls are executed in the scope of the same transaction

2See [http://svn.collab.net/repos/svn/trunk/subversion/libsvn_fs_base/notes/structure](http://svn.collab.net/repos/svn/trunk/subversion/libsvn_fs_base/notes/structure) for the full story

---

40
2.5 The XeR Implementation – xSVN

![xSVN - Storage Layer Diagram]

**Figure 2.12:** xSVN Storage Layer

2. a directory entry list in special format called skel or a delta that reconstructs a directory entry list skel from another directory entry list skel stored in the same table.

3. a property list skel or a delta that reconstructs a property list skel from another property list skel stored in the same table.

From looking at a single strings entry it is impossible to predict what kind of data it represents (a directory, a file, properties or their deltas); SVN employs the representations table for keeping this information. Its entries are links that address entries in the strings table together with information about what kind of strings entry it references, and — if it is a delta — which entry in the strings table this delta is against. Once more, SVN stores only the HEAD revision explicitly (i.e. as a fulltext) in the strings table. Other revisions of any kind of entity (a file, a directory or a property list) are re-computed by recursively applying inverse deltas from the HEAD revision.

To instantiate xSVN from SVN, we had to link the DB XML library into SVN and add a new type of entry in the representations table that points to the HEAD revision of any XML document in the DB XML container (see Figure 2.12). We have decided to use only one DB XML per xSVN repository in order to avoid extra...

---

1. a list of SVN properties that are associated with a directory or a file
2. Containers are entities in DB XML that contain XML documents. In essence, a container is a BDB database, which in turn, just a file on disk that contains all the data associated with XML documents, e.g. metadata or indices.
2. AN XML-ENABLED REPOSITORY

complexity. The container is located in the same folder as other SVN BDB tables are, which automatically enables sharing of the same BDB environment, and in particular transactions.

Since we modified only the storage module of SVN, from the user perspective there is no difference between SVN and xSVN: all the SVN client commands are naturally retained and have the same semantics. However, under the hood when committing XML document the processes are slightly different. Assume that we commit a newly added XML file. Its content is not going to be stored in the strings table anymore. Instead the XML file is added to the DB XML container with a name which is equal to the foreign key stored in the also newly created representations entry of DB XML fulltext type. For each modification in a new commit transaction new representations are created with a unique foreign key that points either to the strings table or to the DB XML container. That foreign key defines the key-mapping function $\kappa$ for all nodes within a repository.

In case an ill-formed XML file is being committed, DB XML fails to accept it and therefore aborts a transaction which is shared within the whole commit. As a result no commit has been made due to its atomicity.

When a working copy is being checked out or updated, xSVN knows what files are stored in DB XML because of the DB XML fulltext representation type, and these files are retrieved from the DB XML container, and not from the BDB strings table.

Modification of an existing XML file is done as follows. The older revision of that file is removed from DB XML, the newest revision is added to DB XML and a delta between these revisions is stored in the strings table which represents an older revision of the XML file in question. This delta has a typical SVN format, and the SVN deltification algorithms are retained unmodified in xSVN. Thus we are still able to receive older revisions of XML documents. For non-XML files the workflow of xSVN is absolutely the same as in SVN: data are stored in the same BDB tables, and the code behaves entirely the same way. Thereby we are also able to store text or binary data in xSVN which can supplement the collection of XML files (e.g. licensing information or generated from XML PDFs). Obviously, XML and non-XML files can be committed in the same transaction.

As was mentioned above, xSVN deltification algorithms are inherited from normal SVN which in turn allowed us to simplify implementation and have consistent deltification regardless of the file type. The natural course of things for the versioned XML data store would be to substitute or extend these algorithms with XML-differencing algorithms (XML-diff). However, the author has decided against this at the xSVN level due to the SVN overall complexity. Additionally, the text-based diff-algorithms are efficient, fast and reliable, nicely fit into the xSVN architecture. Finally, it is not clear whether there is any advantage of changing the deltification in the XeR storage module. However, it is clear that XML differencing brings great advantages in the client both in terms of smaller and less invasive deltas, and more informative conflict resolution.

\footnote{Although, the term of a commit transaction is mostly known from the relational databases domain, in this thesis we are using it with regard to VCSs}
strategies. Therefore we do not renounce XML-diff but rather augment TNTBase with it on a higher level, leaving internal xSVN deltas text-based.

2.6 Conclusion

This chapter serves as a starting point to solving a long-haul XML preservation problem by introducing and elaborating the concept of an XML-enabled repository. Two main theoretical results, Theorems 2.1 and 2.2 show that XeR preserves information about XML documents by storing them in an XML data store. These theorems allowed us to simply instantiate XeR from a versioned control system that follows the described versioning model and an arbitrary XML data store by only realizing a data store function and a key-mapping function.

We have learned that the result called an XML-enabled repository can be built with relatively little efforts by utilizing existing technologies. In particular, the engineering approach consists of several techniques that demonstrate us how to integrate an XML database into a VCS by only posing light limitations on the systems. Additionally, we have looked into several design decisions that should be taken into account when building XeR. But regardless the outcome of those decisions the XeR approach remains unchanged. If for some reason there are no acceptable VCSs with the described versioning model, a developer can still implement his own VCS from scratch since this chapter is general and detailed enough to do that.

Implementation of the XeR approach, xSVN, a part of the TNTBase system, has been obtained by integrating Berkeley DB XML into the Subversion’s storage module and has successfully shown the feasibility and practicality of the approach. But despite that particular implementation decision of employing SVN and DB XML, the XeR approach can be likewise utilized for other VCSs and XML DBs. For example, one possible combination could be an integration of XML DB eXist into a still popular version control system CVS that also conforms to the versioning model introduced in this chapter. We expect only technical challenges to be approached here since all the theory elaborated in this chapter applies to those systems too. Unlike SVN, CVS unfortunately does not have a notion of cheap copying. Therefore one of the biggest challenges would be to extend CVS versioning model with the cheap copying introduced in this chapter. However, cheap copying is not strictly necessary for XML versioning – it just makes copying operation more efficient.

Despite the promising concepts, an XML-enabled repository, and xSVN in particular, is not very useful without an extension since from the user perspective it does not considerably differ from a usual VCS besides that XeR forbids commits of ill-formed XML files. The XeR concept is just an intermediate step towards a feature-rich versioned XML database, and the next Chapter will demonstrate how to take advantage of the XeR model to obtain fine-granular access to XML documents. By moving further in theory as well as in practice we will explore how to obtain a full-fledged XML-database with the versioning aspect.
2. AN XML-ENABLED REPOSITORY
Chapter 3

Towards a Versioned XML Database

Pessimists, were told, look at a glass containing 50% air and 50% water and see it as half empty. Optimists, in contrast, see it as half full. Engineers, of course, understand the glass is twice as big as it needs to be.

Bob Lewis

3.1 Introduction

In the previous chapter we have considered theoretical and engineering aspects of an XML-enabled repository. We made sure that an XML data store of an XeR stores and represents an XML FS sub-tree of the latest repository revision at any given point in time. However, an end user of an XeR still cannot benefit from it since an underlying repository is required in order to obtain a file system information about XML files stored in the XML data store. Additionally, XML data store is not different much from a usual data store with the exception that the former guarantees that it keeps only well-formed/valid XML files. The biggest advantage of XML databases is that they provide fine-grained and efficient access to XML fragments, typically, via the XQuery query language which allows to aggregate information from different sources and present it as an XDM instance. So if an XML data store in an XeR was an XML database that is also aware of the FS structure then an end user would benefit a lot by having versioning and fine-grained access to latest revisions of XML documents at the same time.

In order to achieve this we first have to formalize the concept an XML database and see how it can be utilized in an XeR (Section 3.2). Based on the worked out theoretical foundation we can proceed with analyzing more advanced use cases and build more sophisticated XML-aware services. We will go beyond working with only HEAD
revisions of XML files (see Section 3.3) - and see, for example, how previous revisions of XML documents can be queried and the versioning information can be obtained via introducing XQuery extensions. The resulting toolkit elaborated in this chapter is called a Versioned XML-database (VXD) which consists of an abstract architecture and engineering techniques that help to design a system that goes beyond typical XML databases. Section 3.6 will present a robust implementation of a VXD – the TNTBase system. We will look into technical details that may be important when realizing your own VXD instance. The example implementation may serve as a guideline to plan your own steps towards a full-fledged versioned XML-database with rich functionality.

Before going into details let us first discuss the practical goals that help us to give a better intuition about what we are going to build and why that is needed.

### 3.1.1 Goals

**G1 Querying repository’s XML content:** All XML databases support some sort of querying language for XML. The majority of them implement XQuery W3C specification that became a standard for XML querying. Querying plays an important role in retrieving and aggregating pieces of information. Making this efficient is a key part of any XML database. In this thesis we are not concerned about low-level database optimizations, but we do emphasize that the proposed versioned XML-database toolkit with an integrated XML database $X$ does not sacrifice efficiency of data querying in comparison to pure $X$.

**G2 Modifying XML documents:** XQuery Update [Cha+08] has been designed as a high-level modification language that operates on XML nodes rather than on XML characters.

XML modifications in a VXD must be done in correspondence with the temporal aspect that the versioning concept introduces. XQuery Update modifications can be more concisely expressed and can be applied to multiple documents at the same time.

**G3 Accessing and querying previous revisions:** Modifications on previous revisions must not be allowed since the important principle of any versioning system (including an XeR) is to make history persistent. However, it is clear that accessing and querying XML documents of previous revisions is useful. As an example, a user may want to track modifications of a certain XML node – that is where querying over the full history naturally comes into play.

**G4 Indexing:** One of the major differences between an XML database and a file system as a storage for XML documents is that the former indexes XML nodes in order to speed up data retrieval. In the presented approach all indexing mechanisms are inherited from $X$.

**G5 XML-database views:** Allow user-defined XML-database views whose content is defined by queries. Views are an essential part of many relational databases.
3.2 Versioned XML Database Fundamentals

This concept can also be developed in the XML databases domain. Whereas views may perfectly exist without a versioning aspect, the latter makes them even more widely applicable.

G1-G4 are the items covered in this chapter. Item G5 is discussed in Chapter 4.

3.2 Versioned XML Database Fundamentals

Awareness of the repository infrastructure is a limitation that complicates data access patterns and sacrifices efficiency. By duplication of FS metadata, we can achieve independent and efficient fine-granular access to XML as much as it is allowed by standard XML querying languages.

We are about to develop necessary formalism to achieve:

- Independent exposure of XML HEAD revisions to a client and at the same time preserving history information efficiently.
- Preservation of the XeR cheap copying semantics without sacrificing XML functionality.
- Retention of the XeR interface inherited from the underlying VCS V unmodified for an end user, and at the same time extending the XeR interface with XML-related functionality.

Since XQuery has become a solid standard for querying XML, we will assume that XML databases that we consider implement XQuery specification to a reasonable extent, and will start with the definition of an XQuery processor.

**Definition 3.1 (An XQuery Processor)** Let XQ be a set of strings of the XQuery language. An XQuery processor is a function $\xi : X \times XQ \rightarrow X$ that takes an XDM instance and an XQuery string as arguments and returns another XDM instance.

**Example 3.1 (An XQuery Processor)** Consider the XML document $D$ which is an XDM document node ($D \in X_D$ - a set of XDM document nodes):

```xml
<person type="Doctor">
  <first-name>John</first-name>
  <last-name>Smith</last-name>
  <country>Germany</country>
</person>
```

and an XQuery query $q$:

```xml
<result>
  { 
    fn:string-concat(/@type, " ", /first-name, " ", 
                     /last-name, " lives in " , /country)
  }
</result>
```
3. TOWARDS A VERSIONED XML DATABASE

Then the result of $\xi(D, q)$ will be also an XDM instance:

```
<result>
  Doctor John Smith lives in Germany
</result>
```

There are many standalone XQuery processors like Saxon [Kay08], Zorba [Zor] or XQilla [Sne12]. In the XML community, XML databases simply turn to be XQuery processors with an underlying data store for XML documents. Some databases may utilize existing standalone XQuery processors. For example, Berkeley DB XML [Berb] employs XQilla.

The definition of an XML database has not been standardized so far. However, it is typically implied that an XML DB must have a logical model for an XML document, for instance DOM, the XML Infoset or XDM. Additionally, a logical unit of storage must be an XML document. We will focus on XML databases that utilize the XDM model as it is the most sophisticated one. Also we assume that an XML DB implements XQuery specification defined by W3C since XQuery became an XML database querying language, and all modern XML databases support it, e.g. Berkeley DB XML, eXist [Exi], BaseX [Bas], MarkLogic [Mar], etc.

Based on what is said above, let us fix the notion of an XML database for further referencing. We will make an extensive use of the XQuery language, and will assume that a reader is familiar with it. We will also use standard XQuery functions defined by the W3C specification [W3c] and prefix them with the fixed `fn` namespace prefix.

Definition 3.2 (An XML Database) An XML database is a tuple $\mathcal{XD} = \langle X, \xi, \mathcal{N}, \omega \rangle$ where

1. $X = \langle X, \xi, \mathcal{N}, \omega \rangle$ is an XML data store.
2. $\xi$ is an XQuery processor.
3. $\mathcal{N}$ is a fixed XML database namespace. We also fix an XML database prefix $\text{ns}$ that is bound to the $\mathcal{N}$ namespace.
4. $\omega : X \to \mathcal{U}$ is an injective key-mapping function that maps data store keys to URLs (the set $\mathcal{U}$) whose schema is an XML database prefix $\text{ns}$.
5. For all $k \in X$ such that $\mathcal{C}(k)$ is defined we have $\xi(\text{fn:doc}(\omega(k))) = \mathcal{C}(k)$.

Note, that we omit the first parameter to the $\xi$ function. When we do this we imply that $\xi$ is an XQuery processor of some XML database, and the first parameter is a sequence comprised from the image of the underlying XML data store content function $\mathcal{C}$.

1An image of the XML data store content functions are XDM document nodes, and therefore the sequence of elements in the image is also an XDM instance.
3.2 Versioned XML Database Fundamentals

Example 3.2 (Berkeley DB XML) Berkeley DB XML is an XML database $\mathcal{X}_\delta = \langle \mathcal{X}, \xi, \mathcal{N}, \omega \rangle$ that utilizes Berkeley DB key-value data store as an underlying storage.

Documents in DB XML are identified by their names (keys). The document with name $k$ can be retrieved via the DB XML shell by executing the getDocument command that defines the value of the content function $C(k)$. DB XML uses XQilla as an XQuery processor $\xi$. Additionally, the database namespace $\mathcal{N}$ of DB XML is http://www.sleepycat.com/2002/dbxml and DB XML reserves the XML database prefix dbxml that is bound to $\mathcal{N}$.

The DB XML documentation specifies that an XML document with name $k$ are retrievable from DB XML via an XQuery $q$ of the form:

\begin{verbatim}
fn:doc('dbxml:/<container-name>/k')
\end{verbatim}

The container name is a DB XML-specific notion that groups a collection of XML documents. Let us assume that we have just one container and its name is fixed to container. Then we construct a key-mapping function $\omega$ (see Clause 4 of Definition 3.2) as $\omega(k) = "dbxml:/container/k"$. Then we have $\xi(\omega(k)) = \xi(fn:doc("dbxml:/container/k")) = C(k)$, and therefore property 5 of Definition 3.2 holds. Thus DB XML is indeed an XML database in a sense of Definition 3.2.

Now let us get back to XML-enabled repositories.

Lemma 3.1 Every XML database $\mathcal{X}_\delta = \langle \mathcal{X}, \xi, \mathcal{N}, \omega \rangle$ can be used as an XML data store $\mathcal{X}_\chi$ of the XML-enabled repository $\mathcal{X}$.

Proof: It is sufficient to take $\mathcal{X}_\chi = \mathcal{X}$. \hfill $\square$

Lemma 3.2 If we employ an XML DB $\mathcal{X}_\delta$ as an XML data store of the XML-enabled repository $\mathcal{X}_\chi$, then all XML files of the latest revision are accessible via XQuery fn:doc function of the XQuery processor $\xi$ of the database $\mathcal{X}_\delta$.

Proof: This is a direct corollary of Theorem 2.2 and Definition 3.2. \hfill $\square$

Lemma 3.2 shows that we can already query XML files of the HEAD revision of the XML-enabled repository. Querying is a powerful means to search and aggregate data. Consider the following example.

Example 3.3 (Querying XML-enabled Repository) Assume that we are managing a collection of XML documents of the form presented in Example 3.1 and storing them in the XML-enabled repository. It becomes very easy to get all doctors who live, for instance, in Germany if we employ an XML database as an XML data store for the repository. The desirable results can be obtained by a simple XQuery query:

\begin{verbatim}
fn:collection()[country = "Germany"][@type = "Doctor"]
\end{verbatim}
3. TOWARDS A VERSIONED XML DATABASE

In the meanwhile new people can be added to a repository, or deleted from it. We can always be sure that re-executing the query above will deliver up-to-date results.

From now on we will assume that an XML-enabled repository contains an XML database as its XML data store.

One of the strongest features of an XML-enabled repository is the ability to fetch content without a necessity to have a checked out version of the repository which is a significant advantage especially when the amount of query results is relatively small comparing to the whole content of the repository. Moreover, XML databases have internal knowledge of the XML documents models (typically, backed up by special database indices) that makes information retrieval significantly faster when comparing to sequential search in the file system tree of the checked out repository revision.

Although we already made a significant step forward, the model presented so far has a significant downside. Assume that in our repository people with different professions are stored in XML files that reside in different folders. For instance, teachers are in the teachers folder, accountants are in the accountants folder, etc. It is not possible to query for all teachers that live in Germany given that XML documents do not contain such information. The reason for that is that FS information about document hierarchy is not present in the XML database – it is contained only in the repository FS trees. In order to make it accessible to XQuery processor that information should be associated with every XML document stored in the XML database. The next section is devoted to presenting a way how to enable FS-aware query access to XML documents within XML-enabled repository by slightly augmenting repository operations concerned in Section 2.2.3 that will associate certain metadata to XML documents.

3.2.1 XML-enabled Repository with Metadata

Metadata is information about data describing its certain properties in order to better categorize information or simplify access to it.

We will consider metadata associated with XML documents in an XML database and its underlying XML data store.

Definition 3.3 (A Metadata-enabled Data Store) We call a data store $D = \langle K, C, \mu \rangle$ metadata-enabled if a metadata function $\mu : K \times M \rightarrow \mathbb{B}^*$ is additionally defined. $M$ is the set of metadata keys.

For example, every file on a file system has metadata associated with it, e.g. creation date or access rights: $\mu('/root/text.txt/', 'rights') = '-rw-r--r--$'.

Definition 3.4 (A Metadata-enabled XML Data Store) We call an XML data store $X = \langle K, C, \mu \rangle$ metadata-enabled if it is metadata-enabled in terms of a data store and a metadata function returns instances of XDM: $\mu : K \times M \rightarrow X$.

Based on that we can give the following definition:

\[ \text{Since XQuery is fully specified by W3C and all contemporary XML databases support it, we will use XQuery syntax to define functions where appropriate.} \]
Definition 3.5 (A Metadata-enabled XML Database) A metadata-enabled XML database \( \mathcal{XD} = (\mathcal{X}, \xi, \mathcal{N}, \omega) \) is an XML database, where

1. \( \mathcal{X} = (\mathcal{K}, \mathcal{C}, \mu) \) is a metadata-enabled XML data store.
2. Additionally, an XQuery metadata function is defined in the namespace \( \mathcal{N} \) with the following signature:
   \[
   \text{ns:metadata}($m \text{ as } \text{xs:string}, \$d \text{ as } \text{node()}\text{)} \text{ as } \text{item()}? \]
   such that if \( \mu(k, m) \) is defined then:
   \[
   \xi(\text{ns:metadata}(m, \text{fn:doc}(\omega(k)))) = \mu(k, m)
   \]

Example 3.4 (Metadata Berkeley DB XML) Berkeley DB XML provides a possibility to associate metadata to XML documents with any of the data types specified in the XDM specification. Then this data can be accessed via the XQuery query
\[
\text{dbxml:metadata(document−node(), } \text{xs:string)}
\]
The name of an XML document is actually a metadata value of the metadata key \( \text{dbxml:name} \), therefore a document name can be obtained via XQuery. For example, the name of the first document in the collection can be retrieved in the following way:
\[
\text{dbxml:metadata(’dbxml:name’, fn:collection()[1])}
\]

Before introducing metadata-enabled XML databases we were quite relaxed about posing constraints on the XML database. The only constraints were the underlying XDM data model for storing and accessing XML documents, plus availability of an XQuery processor for querying underlying XML documents. In this case it is hard to come up with examples of XML databases that do not meet these requirements. However, not every XML database directly supports a metadata function. This makes it harder to justify the approach described below. The good news is that metadata function can be easily emulated in any XML database (Appendix B.3 contains a method for achieving this). Therefore without loss of generality we assume that every XML database is metadata-enabled. Later, when mentioning an XML database, we will assume that it is metadata-enabled.

Now we can redefine repository operations so that XQuery queries can access file system information of the committed XML files by considering that an XML database of the repository is also metadata-enabled.

3.2.2 FS-Aware Versioned XML Database

One of the most important theoretical results of the previous chapter is Theorem 2.2. We saw that an XML storage of the XML-enabled repository represents an XML FS-subtree of the checked out version of the HEAD revision. Now let us give a definition what it means for an XML database to represent the same XML FS-subtree. We will reserve an XML namespace \( \mathcal{V} \) for XQuery functions that we define additionally. We will assume that this namespace is bound to the fixed prefix \( \text{vx} \).

Definition 3.6 (An FS-enabled XML-database) We say that an XML-database \( \mathcal{XD} = (\mathcal{X}, \xi, \mathcal{N}, \omega) \) represents an XML FS-subtree \( \mathcal{X}_\mathcal{V} \) of the FS-tree \( \mathcal{T} \) if and only if:
3. TOWARDS A VERSIONED XML DATABASE

(1) $\mathcal{X} = (\mathcal{X}, \mathcal{E})$ represents $\mathcal{X}_T$ with an fs-mapping function $\phi$.

(2) There is an XQuery FS-aware function $\text{vx:doc}$ analogous to $\text{fn:doc}$ with the following signature: $\text{vx:doc(}$ path as xs:string $\text{)}$ as document−node()? as well as an injective path mapping function $\theta : \mathcal{P} \rightarrow \text{xs:string}$ such that for every file node $f \in \mathcal{X}_T$ holds $\mathcal{C}(\phi(f)) = \xi(\text{vx:doc(}$ $\theta(\mathcal{P}(f)))).$ Remember $\mathcal{P} : \mathcal{V}_{\mathcal{X}} \rightarrow \mathbb{N}^*$ is a path function (see Definition 2.4).

This definition gives us a hint what we are heading to: we want to construct an FS-aware function and a path mapping function so that we can query XML documents based on the original file system location. In such a way it would be possible to query a certain XML file instead of the whole collection that itself gives more flexibility and more fine-grained access to XML data. To achieve this we have to extend our repository operations so that certain metadata values are associated with every XML document from a HEAD revision. Remember that an XML-enabled repository uses a metadata enabled XML-database. In particular, that means that a metadata-enabled XML data store is used for storing fulltext revisions of XML documents. Thus when redefining repository operations we just have to additionally define a metadata function.

Going a bit ahead we will add path-related metadata to every XML document. Therefore we need the following definition:

**Definition 3.7 (A Path String and a Path String Function)** A path string (or just an FS path) is a value of the path string function $p : \mathcal{P} \rightarrow \text{xs:string}$ defined as follows. If $\mathcal{P} = \langle n_1, ..., n_m \rangle = n((v_x, v_y))$ is a path from $\mathcal{P}$ then if $v_y$ is a directory node then $p(\mathcal{P}) = "\text{/n}_1/\ldots/\text{/n}_m\text{"}$, and $p(\mathcal{P}) = "\text{/n}_1/\ldots/\text{/n}_m\text{"}$, otherwise.

Example: if we have a document SIGMOD.xml in the folder papers of our repository, then the path string of that document will be /papers/SIGMOD.xml, and a path string of its parent folder is /papers/ (note the trailing slash for directory paths).

Let us now augment operations on the repository $\mathcal{X}$ for creating revision $n$. We will operate with the metadata key vx:path and will associate data of type xs:string* (a sequence of 0 or more strings) that will denote paths that a particular XML file node represents, e.g. in case of copying a single node in the repository can represent multiple paths.

**Addition** For all XML file nodes $f$ that are being added to the XML-enabled repository at revision $n$, we define a metadata function $\mu(\kappa(f), \text{"vx:path"}) = p(p_T)$ where $p_T$ is a path of $f$ in $\mathcal{T}_n^\mathcal{X}$. For all other XML fulltext nodes $\mu$ remains the same.

**Deletion** For all XML file nodes $f$ that are being removed from the HEAD revision, we delete value $p(p_T)$ from the sequence $\mu(\kappa(f), \text{"vx:path"})$ where $p_T$ is a path of $f$ in $\mathcal{T}_{n-1}^\mathcal{X}$. For all other XML fulltext nodes $\mu$ remains the same.
3.2 Versioned XML Database Fundamentals

**Copying** Assume that a node \( d \) is being copied in revision \( n \), and \( f \) is an XML file node child of \( d \) in revision \( n - 1 \). Then by definition of the copy operation there will be a node \( f_c \) in \( T_X^n \) such that \( \zeta(f_c) = \langle \eta(f), n, n - 1 \rangle \) (A copied XML node that share the same node identifier as the node that it was copied from, plus its copy identifier that points to revision \( n - 1 \)). Then we add a new value \( p(P(f_c)) \) to the sequence \( \mu(\kappa(f), "vx:path") \). For all other XML fulltext nodes \( \mu \) remains the same.

**Modification** Assume that a node \( f \) of revision \( n - 1 \) is a target of modification and it is an XML file node. Since its type in revision \( n \) becomes delta, then by Definition 2.22 it cannot be stored in \( X \), therefore we make \( \mu(\kappa(f), "vx:path") \) undefined. Additionally, we have added a new XML file node \( f' \) to \( X \), and we define value of \( \mu(\kappa(f'), "vx:path") \) as in the case of addition. For all other XML fulltext nodes \( \mu \) remains unchanged.

**Definition 3.8 (FS Metadata Field)** An FS Metadata Field of an XML document is a value of function \( \mu(\kappa(f), "vx:path") \), where \( f \) is a corresponding file node in the repository.

We have augmented all operations on an XML-enabled repository that allows us to express the following theorem.

**Theorem 3.1** An XML database \( XD \) of the XML-enabled repository \( X \) with \( n \) revisions represents an XML FS-subtree \( X_C^n \) of \( X \). That is the XML database \( XD \) of the XML-enabled repository contains the last versions of all committed XML documents without losing their file system locations, and, additionally, each HEAD XML document can be accessed via an XQuery expression.

**Proof:** By Theorem 2.2 we already know that \( X_X \) represents \( X_C^n \) with fs-mapping function \( \phi \). So the only thing that is missing is to construct an XQuery function \( vx:doc \) and a path-mapping function \( \theta \) such that \( \xi(\phi(f)) = \xi(vx:doc(\theta(P(f)))) \) for all XML nodes \( f \) from \( X_C^n \). We know that \( f \) has metadata with the key \( vx:path \) associated with it, and its value is \( p(P(f)) \) which is unique in \( XD \) since all paths are different in \( X_C^n \) due to clause 3 of Definition 2.3. Therefore we can select the content of \( f \) via the following XQuery function \( vx:doc \) by providing \( p(P(f)) \) as a function parameter:

```xml
declare function vx:doc($path as xs:string?) as document−node() {  
  if (fn:empty($path)) then  
    () (: an empty sequence if $path is empty:)  
  else  
    let $doc as document−node() := fn:collection()[ns:metadata('vx:path', .) = $path]  
    if (fn:empty($doc)) then  
      fn:error(QName('http://vxd.com/ns', 'NoDoc'), fn:concat('No document: ', $path))  
    else  
      $doc  
  }
```

53
Note that \texttt{vx:doc} throws an error like \texttt{fn:doc} when no document was found. If we take \( \theta(P(f)) = p(P(f)) \) then we obtain the desired result. On one hand, the content of \( f \) is \( \xi(\phi(f)) \), on the other hand it is \( \xi(\texttt{vx:doc}(p(P(f)))) = \xi(\texttt{vx:doc}(\theta(P(f)))) \). Thus the condition \( 2 \) of Definition \( 3.6 \) holds.

The theorem above allows us to query HEAD revision of XML documents by providing their paths. The important advantage of the described approach is that XML file can be retrieved and queried without a need to access repository information – one just needs to know a path of the XML file of interest (recall the signature of \texttt{vx:doc}).

Now we can finally define the notion of a versioned XML-database:

**Definition 3.9 (A Versioned XML-database)** A versioned XML database (VXD) is an XML-enabled repository where metadata repository operations of this section are defined and a metadata-enabled XML database is used for storing HEAD revisions of XML files where the \texttt{vx:doc} XQuery function of the Theorem \( 3.1 \) is defined.

Before introducing the general architecture of a VXD, we have to extend our theoretical foundations with a full-featured file system in \( X \) that is maintained according to the repository modifications and can be efficiently accessed via XQuery.

### 3.2.3 A File System for \( X \)

Naturally, every XeR has a notion of a file system which is reflected in a working copy of the respective repository. We have shown in Section \( 3.2.2 \) that the file system information can also be natively preserved in the underlying XML database \( X \) that results in the possibility of querying XML documents based on their repository path (i.e. a path string introduced in Definition \( 3.7 \) or just an FS path).

Examples of FS paths are: “/”, “/docs/papers”, “/docs/papers/XMLPrague.xml”, “/root.xml”. FS paths that end with “/” we call FS dir paths and they correspond to directories in FS, other paths (that is those that do not end with “/”) we call FS file paths and they, in turn, correspond to the files in the repository FS. FS dir path “/” is called a root FS path or just FS root and, as the name suggests, represents a root of the FS-tree that represents a particular revision in the repository (recall Definition \( 2.20 \)).

The intuition behind an FS-path is very well known from all file systems that users work with. Therefore we will not explicitly define the notions of direct/indirect child/parent FS paths, but rather assume that these concepts have been already defined and well-understood.

However, given the FS metadata model introduced in Section \( 3.2.2 \) it is not easy, for instance, to query for all directories that are contained in some particular folder. What we want to have is a function like \texttt{vx:fs-subfolders($path as xs:string) as xs:string*}. Generally speaking, we want to have an efficient data structure that indexes an FS structure like all contemporary file systems do. While FS metadata fields (see Definition \( 3.8 \)) are sufficient to represent a file system structure in \( X \), this approach has a number of drawbacks:
1. There is no directory entity in $\mathcal{X}$ since FS dir paths are not represented in the FS metadata fields, that makes it harder to obtain information about directories or files within them.

2. Since we do not have a directory entity, we cannot associate additional metadata to directories. One useful example is to be able to tag folders within $\mathcal{X}$.

3. Without an FS index it is not possible to perform more complex query on a file system structure. For instance, get all folders that have 2012 in their name and contain more than 3 XML files. Classical file systems like NTFS of ext3 do not have such a possibility either, but as we will see below it is possible to realize it within $\mathcal{X}$ by utilizing the full power of XQuery.

The main idea that solves the issues above is to have auxiliary XML documents to represent an FS structure of documents in $\mathcal{X}$. We will call such documents FS Documents or FSDs.

First we formally define a concept of a FS document:

**Definition 3.10 (An FS-Document)** Let $\mathcal{X}_C$ be an XML FS-subtree that is represented by a HEAD revision of VXD $\mathcal{X}$, and $d$ is a directory node $d \in D_{\mathcal{X}_C}$. We also fix $\{d_1, ..., d_n\} = \{d_i \in D_{\mathcal{X}_C} | \langle d, d_i \rangle \in E_{\mathcal{X}_C}\}$ and $\{f_1, ..., f_n\} = \{f_i \in F_{\mathcal{X}_C} | \langle d, f_i \rangle \in E_{\mathcal{X}_C}\}$.

An FS document $\text{fsd}(d) \in \mathcal{X}$ is an XDM document node:

```xml
<vx:fsd path="p(\text{p}(d))">
  <vx:dir name="n((d, d_1))"/>
  ...
  <vx:dir name="n((d, d_n))"/>
  ...
  <vx:file name="n((f, f_1))"/>
  ...
  <vx:file name="n((f, f_n))"/>
</vx:fsd>
```

An example of an FSD for directory /docs/ may look like the following:

**Listing 3.1: Example of an FSD**

```xml
<vx:fsd path="/docs/">
  <vx:dir name="papers"/>
  <vx:dir name="presentations"/>
  <vx:file name="README.xml"/>
  <vx:file name="LICENSE.xml"/>
</vx:fsd>
```
3. TOWARDS A VERSIONED XML DATABASE

Note that non-XML files are not listed in FSDs since $\mathcal{X}$ stores only XML content, which is another way of seeing that $\mathcal{X}$ represents an XML FS-subtree of the checked out HEAD revision of $\mathfrak{VX}$.

**Definition 3.11 (A VXD File System)** A file system of VXD $\mathfrak{VX}$ is a set $\mathcal{F}_{\mathfrak{VX}} = \{ \text{fsd}(d) | d \in D_{\mathfrak{VX}} \}$

Additionally, FSDs give us enough freedom to introduce additional FS system entities like symbolic links or virtual documents discussed in Chapter 4. Moreover, due to flexible structure of XML file system entities can be augmented with additional meta-information like a type of an XML document or the total size of a directory. FSDs are stored in the underlying XML-database $\mathcal{X}$ and therefore automatically amenable to querying and enable efficient retrieval of FS-related information (Section 3.2.3.2). However, in order to benefit from it, we must ensure that they always represent the FS index of the HEAD revision file system structure. This is a topic of the next section.

### 3.2.3.1 Managing FSDs in XeR

In order to benefit from FSDs, their content should obviously represent the latest state of HEAD revision FS structure. Thus FSDs must be managed whenever a new revision appears in the underlying repository.

Since FSDs are XML documents and are stored in an XML-database, the natural way of modifying them is to utilize XQuery Update facilities provided by $\mathcal{X}$.

Let us define algorithms of managing FSDs in the XQuery pseudo-language.

**Addition** Assume we add an FS-tree $T$ with a root node $v_r$ to the directory node $d$ in the repository $\mathfrak{R}$ with an entry name $N \in \mathbb{N}$.

```xml
declare updating function vx:fsd-add($r \in D_T$) {
  insert nodes
document {
    <vx:fsd path='fn:concat(p(p(\mathfrak{P}(\mathfrak{R}, d)), /, p(p(r))))'>
      { for $\langle r, v \rangle$ in $\{ \langle r, v \rangle | (r, v) \in E_T \}$ return
        if ($v \in D_T$) then
          (\{<vx:dir name='n(\langle r, v \rangle)'/>\}, vx:fsd-add(v))
        else if $f(v) = \mathcal{X}$
          (\{<vx:file name='n(\langle r, v \rangle)'/>\})
        else ()
    </vx:fsd>
  } into $\mathcal{X}$
};

if ($v_r \in D_T$)
3.2 Versioned XML Database Fundamentals

Essentially, we add an item to the $d$‘s FSD and create FSDs for newly added folders recursively. Note that only XML files are added as child file elements of FSDs.

**Deletion** Deletion is opposite to addition — we have to remove FSDs from a file system. Assume that an entry with the name $N$ is removed from the $R$’s $d$ node.

```
declare updating function vx:fsd-del($v \in V_R$) {
  if ($v \in D_R$) then
    (delete $fsd(v)$ from $X$,
     for $d$ in \{ $d$ | $\langle v, d \rangle \in E_R$ \} return vx:fsd-del($d$))
  else
    ()
};
```

(``delete nodes $fsd(d)$/vx::*[@name = $N$], vx:fsd-del($v_r$)``)

**Copying** The copying operation with regard to FSDs is handled analogously to the addition case.

**Change of an XML file** Since the file system structure remains unchanged, nothing has to be done with respect to FSDs.

Having an up-to-date file system information is highly important. It can be utilized for enabling FS-aware database queries in an efficient and flexible manner.

### 3.2.3.2 Querying through FS

Since FSDs are also XML documents stored in $X$, they are automatically amenable to querying. By Definition 3.10 every FSD document can be addressed by the following XQuery function:

```
declare function vx:fsd($path$ as xs:string*) as document()* {
  if (empty($path$)) then
    () (: an empty sequence if $path$ is empty:)
  else
    let $fsd$ as document-node()? := fn:collection()/$x$://vx:fsd[@path = $path$]
    if (empty($fsd$)) then
      error(QName('http://additional.namespace/vx', 'NoFSD'),
        concat('No directory path exists: ', $path$))
    else
      $fsd$
};
```
3. TOWARDS A VERSIONED XML DATABASE

That is it. Knowing how to access an FSD leaves enough freedom to querying utilizing an \(\mathcal{X}\)'s XQuery engine. Let us provide a few examples to fortify our understanding of the FSD concept.

**Example 3.5 (XQuery and FSDs)** Omitting error handling the XQuery query for listing full file paths in some directory can look like the following:

```xquery
declare function local:full-paths($dir as xs:string) as xs:string {
  vx:fsd($dir)/vx:file/concat($dir, .)
};

local:full-paths(<any directory path>)
```

We can also easily count a number of directories that contain at least \(n\) XML files:

```xquery
fn:count(fn:collection()//vx:fsd[count(vx:file) > n])
```

Returning distinct XML root element names from a backup directory and its subfolders becomes simple too:

```xquery
fn:distinct-values(fn:collection()//vx:fsd[fn:ends-with(@path, "/backup/")]/local-name(.))
```

As a conclusion, FSDs offer a flexible way of natively representing a file system structure of XML documents in \(\mathcal{X}\) with possibly of extending the list of FS entities additionally to files or directories. The important part here is that being XML documents, FSDs are amenable to native querying in \(\mathcal{X}\) in a natural way.

### 3.2.4 Summary

By now we have worked out all necessary formalism to ensure that if we define repository operations in the described way than we can access HEAD XML files both by the check out operation and XQuery expressions. The next sections will show what is still lacking for implementing a versioned XML database in practice.

### 3.3 A Versioned XML-Database Toolkit

In this section we are going to present a versioned XML-database toolkit - the toolkit which parts constitute a versioned XML-database. Based on the theory above Figure 3.1 represents a general architecture of a VXD. It has a VXD interface that serves as an end-point for VXD users or applications that employ a VXD as a component for managing XML documents. A VXD interface makes use of two sub-components: \(x\)Accessor - a component for dealing with XML content, and Vcs\(x\)ccessor - a component that exposes an interface of the underlying XeR. These components can interact with each other in order to obtain supplementary information, e.g. retrieve an XML node together with the repository revision it belongs to. **XQuery Library (XQL)** is a set of XQuery modules that are specific to the VXD implementation and interpret the raw content stored in the underlying XML-database. Last but not least, as we already
saw in Section 3.2.3, we have extended a model of XML documents in $\mathcal{X}$ by introducing a “native” file system in $\mathcal{X}$ and defining File System Documents (FSDs) that represent a file system index.

Let us focus on the toolkit components in more detail. We are going to discuss these components from bottom to top demonstrating how the upper components make use of the lower ones.

### 3.3.1 An XQuery Library

As we have already mentioned, an XQL provides high-level XQuery functions for abstracting from the details of storing and accessing XML documents and information about them (e.g. FSDs). We have already discussed $\texttt{vx:doc}$ (defined in Theorem 3.1) and $\texttt{vx:fsd}$ functions crucial for a VXD. Other functions that are useful for VXD common use cases can be provided additionally. For example, gathering statistics about frequency of XML elements or finding certain information like definition of mathematical concepts in OMDoc documents. Providing convenient data access functions can simplify VXD utilization for the end users, and an XQL component leaves enough freedom to predefine common data access patterns in the form of XQuery modules and functions in them.

### 3.3.2 xAccessor and VcsAccessor

One level up are two components that interact with each other: xAccessor and VcsAccessor. xAccessor is an abstraction over $\mathcal{X}$’s interfaces that mediates requests from a user to an XQL. Additionally, xAccessor is responsible for transforming database query results into data structures of the target language the VXD toolkit is implemented it. For instance, a particular implementation of xAccessor in Java, would transform XDM instances retrieved by an underlying XQuery processor to the Java DOM objects.
3. TOWARDS A VERSIONED XML DATABASE

that are suitable for further processing. VcsAccessor is a component that is responsible for bridging VCS functionality of an XML-enabled repository with either xAccessor or the VXD interface. This information includes change logs, history data, differences between revisions, etc., i.e. VcsAccessor is an embedded VCS client of V. For example, VcsAccessor in TNTBase makes use of the SVNKit library \( \text{SVN} \) - a java client for SVN (more details in Section 3.6).

We can distinguish several most prominent applications of VcsAccessor to give a better idea why it is needed in the first place:

- Retrieve file/folder contents based on their XeR path and a revision number (including non-XML content).
- Get revision information. For instance, retrieve the numerical value of the HEAD revision, or revision numbers which concern a certain document.
- Committing files without a need to have a working copy.
- Retrieve change logs, e.g. for data change analysis.

As discussed before an XQuery processor is a crucial component of any XML-database. As we have already seen the VXD toolkit provides a means to query HEAD revision documents and file system information about them. But since we have much more elaborated persistent layer in XeR, it is obviously beneficial to expose additional XeR information provided by VcsAccessor to an XQuery processor. There are several important use cases that can be enabled by that.

(i) Having an access to any revision of non-XML data from XQuery. For instance an underlying repository may contain JSON files that are converted to XML by XQuery on the fly. Other text-based formats can also be conveniently parsed within XQuery, like CSV or HTML which is not necessarily well-formed XML. Of course, XQuery modules with such functionality can be stored in a VXD itself, and accessed by referencing their repository path.

(ii) Querying previous revisions of XML documents. Here we loosen our initial design decision to be able query only HEAD revision XML documents.

(iii) Being able to query for change logs and differences for any kind of data stored in the repository.

In order to make an underlying XQuery processor leverage the VcsAccessor functionality, a notion of XQuery external function must be utilized that is supported by almost all existing XML-databases. The idea of external functions is to bind a body-less XQuery function declaration to an implementation in the target language. The types of XQuery external function in the scope of a VXD can be found in Appendix B.6.

We will cover in detail items (i) and (ii) since in our experience they bring most of additional value.
3.3 A Versioned XML-Database Toolkit

3.3.2.1 Accessing XeR History via XQuery

Given the notion of XQuery external functions it becomes relatively easy to achieve this. We need to define a signature of an XQuery function and bind it to the implementation in the target language available in VcsAccessor. Leaving out the implementation details in the target language and database-specific binding mechanisms, the signature of the function that returns contents of any file given its path and a revision (we will use vxe as a namespace prefix for XQuery external function):

\[
\text{DECLARE FUNCTION} \ vxe:file-contents(\$path \text{ as } \text{xs:string}?, \$revision \text{ as } \text{xs:integer}? ) \quad \text{AS} \ \text{xs:string}? \ \text{EXTERNAL};
\]

Errors in external functions are typically communicated via exceptions in the target language, e.g. when a path does not exist in the repository.

3.3.2.2 Accessing and Querying Previous Revisions

Accessing previous revisions of XML documents can be done in the same fashion as for any other documents as was covered in the previous sections. In order to query the content of older XML documents, the external function must return an XDM document node instead of just a string. An implementation of the external function must be slightly changed to return a document node:

\[
\text{DECLARE FUNCTION} \ vxe:doc-ext(\$path \text{ as } \text{xs:string}?, \$revision \text{ as } \text{xs:integer}? ) \quad \text{AS} \ \text{document}? \ \text{EXTERNAL};
\]

This function is a natural candidate to be included in an XQL. However, this approach has a number of technical limitations: inefficiency by communicating between layers, necessity to parse the whole document upon retrieval via VcsAccessor and inability to leverage databases’ indices. These issues can be mitigated by caching previous revisions of a subset of XML files in X directly – xAccessor takes care of that. In order to efficiently determine which file nodes are XML file nodes for a particular revision in XeR, a versioning data model introduced in Chapter 2 must be extended to preserve a value of a data store function for each node when it had the fulltext type. Since previous revisions are immutable we do not have to deal with keeping the cache up-to-date. FSDs for previous revisions can be created as well utilizing techniques described in Section 3.2.3.1.

In order to address previous revisions cache in X we introduce a revision metadata field additionally to the FS metadata field (Definition 3.8). Then the optimized XQuery function for XML documents retrieval looks like the following:

\[
\text{DECLARE FUNCTION} \ vx:doc(\$path \text{ as } \text{xs:string}?, \$revision \text{ as } \text{xs:integer}? ) \quad \text{AS} \ \text{document}? \{ \\
\text{LET} \ \$doc := \text{fn:collection()}[\text{ns:metadata(‘vx:path’, .) = } \$\text{path}] \\
[\text{ns:metadata(‘vx:revision’, .) = } \$\text{revision}] \\
\text{IF (fn:empty(\$doc)) THEN} \\
\text{LET} \ \$doc := \text{vxe:doc-ext(\$path, \$revision)} \text{RETURN} \\
(\text{vxe:cache}(\$doc), (: \text{Optionally cache the document (an empty sequence is returned) :)})
\]

51
3. TOWARDS A VERSIONED XML DATABASE

To summarize the advantages of caching previous revisions in $\mathcal{X}$:

- Disk space is spent only for those parts and only for those revisions that are indeed needed for querying.
- Being in $\mathcal{X}$, previous revisions are subject for indexing, and therefore querying time might be improved considerably.

Additional cache eviction strategies like Least Recently Used (LRU) can be employed to solve the problem of constantly growing cache.

Whereas the VXD toolkit was not designed for the efficient access of previous revisions via an XQuery engine, the working solution exists in practice. In our experiments it was far more important to have an efficient fine-grained access to the latest XML documents whereas previous revisions were infrequently demanded.

3.3.3 A VXD Interface

The topmost component of the VXD toolkit is a VXD interface that binds together the functionality of xAccessor and VcsAccessor transitively exposing XML-related functionality spun off from $\mathcal{X}$ and $\mathcal{V}$’s interface. We are not going to tie ourselves to any particular communication protocol or architectural style. It can be REST [Fie00], WebDAV [Web], XML-RPC [Xmlb], SOAP [Box+00], etc. It depends on the user needs and the VXD toolkit architecture is generic enough to leave such freedom to the toolkit developers. In Appendix B.5 we provide some discussion what groups of requests the REST VXD interface may have according to our experiments and experience gained from them.

3.4 Open Problems and Directions to Solve Them

In the previous section we have discussed the general architecture of the VXD toolkit that implements a notion of a versioned XML database with several engineering improvements. However, there are still a few open problems that we have not thoroughly explored yet:

- Updating XML documents via XQuery update must respect history of XML files. Discussion on this topic can be found in Appendix B.4.

- Permission model for documents store in XeR are naturally inherited from $\mathcal{V}$. Exposing access to XML documents via XQuery raises the problem of unauthorized access to certain documents. Our paper [Gra+11] describes a possible solution to that problem. However, some work has to be done in order to integrate it into the VXD concept.
3.5 Related Work

3.5.1 A Time Machine for XML

The work described in \cite{Fou+10} considers a number of interesting and promising approaches for XML versioning which add a versioning model to a database and extend the XPath data model (XDM) to address temporal aspects. Unfortunately, the work seems to be only theoretical, since at the time of writing no implementation was available that proves the concept and we have not seen any updates publicly available from 2010. Furthermore, the mentioned technical report left many open questions that hamper the author from fully evaluating the approach. Moreover, every version of a document is stored as fulltext thus consuming extra space, especially when a document is modified insignificantly. Also the presented work does not support branching and tagging (cf. \cite{Tic85}) which are essential features of contemporary VCSs. Additionally, there is no notion of a working copy which is crucial for collaborative managing of document collections. Last but not least, there is no means to index XML data, which results in the efficiency problems when the stored data set grows. The XeR approach implies that indexing capabilities are inherited from a DB XML \( X \) that brings additional value to our approach.

Taking into consideration all of these drawbacks, it makes the mentioned approach not suitable for real-world scenarios, and for our particular needs inspired by OMDoc use cases.

3.5.2 xDB Versioning

Documentum xDB \( \text{Xdb} \) is a native XML database developed by EMC. It supports basic versioning via an incremental approach: xDB stores only the HEAD revision of XML documents as fulltext, and all others as reverse deltas against newer ones. This approach is very similar to what most VCSs have (e.g. SVN). However, xDB does not support collaborative editing (like an SVN client) – once a file is checked out by one user, it cannot be modified by another, since the file is locked until the first user commits it. This limitation is a very major one for the systems that have multiple users working on the same content at the same time. Additionally, xDB does not support tagging/branching which further limits versioning usability. Last but not least, xDB was not designed for storing and versioning non-XML content like text files or CSS stylesheets. Although, Documentum xDB supports advanced XML versioning compared to many other XML-native databases, described drawbacks pose significant limitations on the document workflows.

3.5.3 DeepFS

The idea of having a file system index stored inside an XML database and thus amenable to querying has been also described and implemented in the scope of the DeepFS project \cite{HGS09}. More precisely, only metadata associated with each file on the file
system are preserved in the XML database. These metadata can be queried and modified, using XQuery/XQuery Update, in particular. Whereas the DeepFS work deals rather with physical file systems, the VXD FS documents propose a similar approach within a versioned storage and for XML documents only.

3.6 Implementation – TNTBase

The TNTBase system is an implementation of the VXD toolkit. We keep this section relatively short since most of the necessary engineering concepts have been already discussed in this chapter. Some of the implementation decisions have been ensued from the xSVN design (see Section 2.5). On the other hand, some decisions have been driven by the use cases of the OMDoc environment. These use cases are described in detail in Chapter 7.

First technical paper about TNTBase has been published in [ZK09] which still remains up-to-date regarding core technical details. However, since then an implementation matured considerably and is being used as a storage for different XML documents (see Chapter 7).

Figure 3.2 depicts the concrete realization of TNTBase based on the general architecture presented on Figure 3.1. On the left hand side we see xSVN described in the previous chapter. The component on the right hand side (that exposes xAccessor and VcsAccessor features) is a stand-alone web service written in Java, and provides a REST interface based on the JAX-RS (JSR 311) Reference Implementation called Jersey [Jer]. An implementation of VcsAccessor called SVNAccessor is based on the open source SVNKit [Svn] library – a Java implementation of the SVN client.

Other relevant details are:

- The TNTBase namespace is http://tntbase.mathweb.org/ns and is bound to the prefix tnt.
- The XQL of TNTBase contains a number of additional functions for operating with a repository, e.g. committing XML documents directly from XQuery statements. The full list of XQuery functions available for an end-user is provided at Appendix C. External functions have the prefix tnte.
- Caching of previous revisions is done in the way fully controlled by a user which was sufficient for our use cases.
- Apart from files and directories FSDs are extended with a third type of a file system entity called virtual documents (see Chapter 4).

Additional information about TNTBase interface can be found at the TNTBase home page [Tnt].

Before concluding the section let us be more specific about one particular implementation problem that concerns consistency.
3.6.1 Consistency of TNTBase FS

So far we have neglected this question due to its implementation specificity. The purpose of this section is to demonstrate challenges that should be expected when realizing a VXD.

FS in a VXD has been carefully discussed in Section 3.2.3. We use XQuery Update in order to manage FSDs. Obviously, those changes should be done in correspondence with changes in FS itself (like file addition or deletion) to keep it consistent. Generally, transactions allow to group operations and treat them as an atomic operation. However, in xSVN (and in BDB-based SVN repositories) there are two different types of transactions:

- **Commit transaction** The global transaction which is initiated when a user commits a number of documents. Either everything is committed or nothing. If a network failure occurs when only a part of files has been sent to a repository then at the end no files will be committed. However, part of them will remain in the repository invisible for a user because of the BDB transactions covered next. These failed commit transactions are called *dead transactions* and require administrative efforts to manually remove them. They do not do any harm except occupying extra space in the repository. Thus only when xSVN finalizes a commit transaction (also via a smaller BDB transaction) then a new revision appears in the repository.

- **BDB transactions** Those are used for supporting consistency of smaller operations in the scope of a *commit transaction*. For example, adding each file to BDB
3. TOWARDS A VERSIONED XML DATABASE

or DB XML is embodied into a single BDB transaction that apart from putting content into a storage also adds additional information to BDB like entities that characterize changes to a repository FS. Putting the whole commit process into one BDB transaction is not practically feasible since a large set of files requires a considerable amount of resources (like RAM) for supporting transaction promises (ACID).

Due to the reasons above, modifications to the TNTBase file system cannot be done upon modification operation itself since dead transactions may lead to the persisting of the intermediate state of the TNTBase file system.

The solution that the author has implemented is to update a FS in a batching mode from a BDB transaction that finalizes the commit transaction. xSVN has a special BDB table changes that accumulates all changes made to xSVN FS during multiple commit transactions that are uniquely identified by a transaction identifier. Given a transaction identifier we can extract FS changes for the current commit transaction. Knowing those changes we can apply algorithms described in Section 3.2.3.1. Even if a failure occurs over those operations then due to the BDB transaction atomicity neither a commit transaction will finalized nor TNTBase FS will be modified.

3.7 Conclusion

This chapter concludes the part devoted to the XML storage. It shows how to approach XML versioning and XML fine-grained access via existing stack of technologies. Theoretical foundations give us the possibility to prove that the presented concept of the versioned XML database indeed works in practice. The described engineering techniques exhibit a practical path from the formalism to the robust system implementation. This path is fortified by the stable implementation of the VXD toolkit called TNTBase.

The strength of the VXD toolkit consists of the possibility to adopt an existing stack of technologies in the XML and VCS fields which becomes crucial when realizing your own VXD. On the other hand, the architecture is general enough to be implemented from the very beginning and can serve as a roadmap for the VXD design despite the VXD toolkit leaves enough freedom to extend a database for a particular XML field. Chapter 6 on the example of TNTBase will demonstrate how a VXD concept can be handcrafted to the particular XML language domain (we will use OMDoc throughout that chapter).

We have not described many practical obstacles that were on our way in implementing TNTBase. Most important of them are described in Appendix B some of which we have already mentioned. Several open problems still exist and have been briefly summarized in Section 3.4.
Part III

XML Workflows
An efficient and flexible versioned XML storage is only a part of the story to make working with XML more hassle-free. The next chapters will deal with the problem of XML workflows, namely flexible XML database views, XML scripting and templating. Being able to work with XML in a more efficient way and relieve VXD users to deal with the technical details they do not have to, is certainly of considerable value.
Chapter 4

Scripting XML Documents

The good thing about reinventing the wheel is that you can get a round one.

Douglas Crockford about “reinventing XML”

4.1 Introduction

One of the big promises of XML as a representation paradigm is that documents become uniformly machine-processable. Indeed XSLT is widely used for pre/postprocessing XML-encoded documents, and XQuery is poised to become for semi-structured data what SQL is for relational data. But in both cases, traditional workflows have important features that are largely missing from XML workflows. (i) Document authoring and management systems allow user-definable, in-document macros that allow the computation repetitive writing tasks or processing of outside data. (ii) Relational databases support database views as first-class citizens, i.e. computational devices that look like tables to the user, but internally are embedded queries. Both in-text macros and views could in principle be externalized from production workflows at the cost of losing locality and ease-of-use. And indeed their integrated nature has brought levels of customization and functionality that have not been achieved in practice without.

In this chapter we present Virtual Documents (VDocs), a general framework for integrating XQuery queries into XML documents as computational devices and processing them efficiently. Although we are focusing exactly on XQuery as means for scripting XML documents, we are not bound to it: we might use theoretically XSLT or XProc for similar purposes.

\footnote{We take \TeX/\LaTeX as the most prominent example from which we take our intuitions. Wikis usually also allow in-text macros and arguably the VB/VBA extensions of Office suites also allow macros, even if they are less extensively used.}
4. SCRIPTING XML DOCUMENTS

As a rough approximation, VDocs are “XML database views” analogous to views in relational databases; these are tables that are virtual in the sense that they are the results of SQL queries computed on demand from the explicitly represented database tables. Similarly, Virtual Documents are the results of XQuery queries computed on demand from the XML files explicitly represented in an XML DB, presented to the user as entities (files) in the database file system. Like views in relational databases VDocs may be editable if they satisfy certain criteria (see Section 4.2.6). In this case an XML database is supposed to transparently patch the differences into original files in its underlying storage. Therefore a user does not have to know about the original source of document parts and it allows him to focus only on relevant pieces of information. Thus like relational database views, VDocs become very useful abstractions in the interaction with a (versioned) XML storage.

We have already discussed VDocs in [ZKR], concentrating on practical and theoretical aspects. In this chapter we present the current state of the art regarding VDocs and discuss its implementation in TNTBase that has been extended and matured considerably. Although a VXD concept presented in previous chapters is strongly tied with a versioning model, the concept of virtual documents is more abstract and can be implemented for any storage model regardless whether it is a VXD or a local file system. However, versioning capabilities of the underlying system make the VDocs concept even stronger.

4.1.1 Database Views

Even though it is typically clear what is meant by a database view, the author is not aware of any its rigorous definition. So let us fix our understanding of a database view that is general enough to cover our use cases:

**Definition 4.1 (A Database View)** A view is a virtual database entity satisfying the following characteristics:

1. Views are closed in the model of the database.
2. Views are the first class citizens in the database.
3. A language for defining views is a query language of the underlying database.

**Example 4.1 (A Relational Database View)** Since data in relational databases are represented as tables, relational database views are also tables (Item 1). Item 2 tells us that from just looking at the data in some table we cannot distinguish a view from a physical table in the database. Finally, since a query language for relational databases is SQL, views are defined by SQL expressions too. As we can see our definition of a database view is in correspondence with relational database views.

**Example 4.2 (An XML Database View)** In this case Item 1 means that if a data model of an XML database is XDM, then views are also instances of XDM. Item 2 tells us that from looking at the retrieved document it should not be possible to distinguish it
from the physical document residing in the database. In our case it means, that views should be a part of the VXD file system (cf. Section 3.2.3) and must be accessible via XQuery expression vx:doc, and therefore be amenable to querying as well. Finally, Item 3 suggests that for an XML database supporting XQuery, the language for defining views must also be XQuery. That also entails that views can be part of other views.

Additionally, views can further be:

- **Read-only or updatable.** Updatable views can be edited like a normal document and the modified parts can be propagated back to the database to their origins. However, it is not possible in all cases.

- **Materialized,** i.e. for faster retrieval of views, their contents can be “cached” in the database. The major problem here is the problem of updating materialized views when the underlying data have been changed and affect the contents of the view.

Our objective is to create a concept of XML database views that satisfies the definition above. An additional goal is to extend this concept to the concept of virtual documents so that views can be defined and maintained more easily.

The following incentives have driven the development of the VDocs concept.

**Separating responsibilities** Separating static parts of XML documents from the dynamic parts makes documents easier to develop and maintain. The analogy of web-designers and programmers may be appropriate here – web-designers are responsible for static HTML parts together with stylesheets whereas programmers are responsible for implementing business logic that generates the dynamic parts of a web-page.

**Improving reusability and extensibility** In many scenarios of document authoring, many tasks are very similar to each other. The output can be completely different but the logics behind computations remains the same – e.g. only input parameters are changed. Thus we want to have means of defining the tasks repetitive in nature and not to lose the generality in the common sense.

**Flexibility** Documents obtained by means of VDocs should not only present a view over XML data, but be parametrizable, searchable, editable and also have the ability to comprise another views.

The last but not least important contribution of this chapter is to support updatable (editable) XML database views without a necessity to hook into the XML-database engine. Plain XQuery with external functions must be sufficient to support updatable views.

Let us start with defining the fundamentals.
4. SCRIPTING XML DOCUMENTS

4.2 Virtual Document Fundamentals

First we specialize Definition 4.1 to the XML workflows case:

**Definition 4.2 (An XML Database View)** Let $\mathcal{XD} = (X, \xi, \mathcal{N}, \omega)$ be an FS-enabled XML-database and $q$ an XQuery query, then we call a pair $\nu = \langle q, p \rangle$ where:

1. $p$ of type $\text{xs:string}$ is a **view path** that uniquely associates with a query $q$
2. $\xi(\text{vx:doc}(p)) = \xi(q)$

an **XML database view** and the corresponding query $q$ an **XML view definition**.

**Lemma 4.1** The XML database view is a database view in a sense of Definition 4.1.

**Proof:** Since an XML database view is defined as a result of some XQuery expression, the former is an instance of XDM, which is a model of an FS-enabled XML-database. Since views are accessible via the $\text{vx:doc}$ function that also serves for retrieving physical documents, it makes the former indistinguishable from the latter. Finally, XML views are defined by XQuery which is a language of the underlying XML database. \[\square\]

Although we defined the $\text{vx:doc}$ function only for versioned XML databases, in this chapter we do not require a database $\mathcal{XD}$ to be versioned. We assume that $\text{vx:doc}$ just returns a physical document or a view based on its file system path (see Definition 3.7) in the database.

**Example 4.3 (View on Doctors)** Recall the documents that describe persons from Example 3.1.

```xml
<person type="Doctor">
  <first-name>John</first-name>
  <last-name>Smith</last-name>
  <country>Germany</country>
</person>
```

We can define an XML database view with a view path /views/doctors.view with the following XQuery query:

```xquery
fn:parse-string(fn:join(vx:collection()/*/[@type="Doctor"]/last-name, ', '))
```

The XQuery above returns a list of doctors’ last names concatenated with a comma, e.g. Smith, Müller, Ivanov, ... Note that the content of the view is not an XML document in this particular example, but of type $\text{xs:string}$ which is also from XDM. This view is accessible via the XQuery function $\text{vx:doc}$ likewise all other XML documents within the XML database.

---

1 We can talk about paths since $\mathcal{XD}$ is FS-enabled, i.e. it represents some FS-tree.
4.2 Virtual Document Fundamentals

However, defining an XML database view (or just a view) via a single XQuery expression is often not the most convenient way due to the number of reasons:

- View queries can be quite big although they just contain lots of XML element constructors and very little computational logic. For example:

  ```xml
  <root type="bar">
    <nested-element-a>
      <nested-element-b>
        \{fn:count(vx:collection())\}
      </nested-element-b>
    </nested-element-a>
    ...
  </root>
  ```

  From the example above it becomes clear that separation between static and dynamic (computational) parts can be made more explicit.

- Users that are not familiar with XQuery still should be able to design and create views or their templates by reusing existing XQuery queries and embedding them into the view definition.

- Having just a single XQuery as a view definition prevents from reusing of “static” parts of that XQuery and the “computational” parts. For instance, in the example above the static parts are all XML elements, and the computational part is responsible for counting number of the documents in the database. For instance, it is desirable to be able to easily substitute the computational part to doubled count without rewriting the whole XQuery.

All the reasons above induced us to come up with a simplified but not less powerful way to define XML database views which we will call Virtual Documents from now on to stress out the difference. Section 4.2.5 will prove that VDocs are indeed XML database views.

### 4.2.1 Defining Virtual Documents

Instead of pure XQuery queries we define Virtual Documents by the virtual document specifications (VDoc Specs) that are explained below. In a nutshell, these are just XML documents with XQuery queries or links to them (embedded as text). Section 4.2.5 will show that VDoc Specs are equivalent to XQuery queries, and therefore VDoc Specs can serve as XML-database views definitions.

When designing the syntax for VDoc Specs the following has been kept in mind:

1. It is still must be possible to define a VDoc with a single XQuery when it is either more convenient or the only way to achieve certain results.

2. VDoc Specs must have an unambiguous translation into XQuery so that the promise of a view being defined by a querying language of the underlying data store is kept.
4. SCRIPTING XML DOCUMENTS

(3) VDoc Specs must facilitate reusability of existing parts of other VDoc Specs and existing XQuery modules.

(4) VDoc Specs must facilitate division of responsibilities.

**Definition 4.3 (A Virtual Document Specification)** A Virtual Document Specification is an instance of the XML language defined by the RELAX NG [Rel] schema (also available at [ZK]):

**Listing 4.1: RELAX NG Schema for VDoc Specs**

```xml
default namespace vx = "http://vxd.com/ns"

#common definitions
id.attrib = attribute xml:id {xsd:string}
name.attrib = attribute name {text}
query.attrib = attribute query {text}
href.attrib = attribute href {xsd:string {minLength = "1"}}
any.attrib = attribute * {xsd:string}
any.el = element * {(any.attrib*, any.el*) & text*}

#query elements – inside the skeleton and outside
query.outside.el = element query {name.attrib, (xsd:string {minLength = "1"} | href.attrib)}
query.inside.el = element query {name.attrib | xsd:string {minLength = "1"} | href.attrib}

#root element of VD specification
virtualdocument = element virtualdocument {id.attrib?, virtualdocument.model}
virtualdocument.model = skeleton, (query.outside.el* & params?)

#skeleton element – either empty with reference to another skeleton or with ”meat”
skeleton = element skeleton {id.attrib? & (skeleton.model | href.attrib)}
#the skeleton should contain exactly one non VXD element as a child
skeleton.model = element * − vx: * {xqel* & any.attrib* & text*}

#typical element of a skeleton – either vx:xqinclude or arbitrary non−VXD element ot text
xqel = xq | element * − vx:* {xqel* & any.attrib* & text*}

#vx:xqinclude element – should contain a query as an attribute, or have a child vx:query
#Query as element should reference a query or contain a query as a text
#The second allowed element is vx:return
xq = element xqinclude {(query.attrib | query.inside.el), return.el}

#vx:return element and its children
return.el = element return {return.internal.el*}
return.internal.el = result.el | text* | element * − vx:* {return.internal.el* & any.attrib*}
result.el = element result {empty}

#parameters
params = element params {param+}
param = element param{name.attrib, value+}
```
4.2 Virtual Document Fundamentals

value = element value {xsd:string {minLength = "1"}}

#virtualdocument is the root element
start = virtualdocument

The RELAX NG schema above gives a formal definition of a VDoc Spec, but, unfortunately, is not easy to read and does not give immediate sense of what VDoc Specs look like. Therefore we are going to start explaining the syntax with an elaborated example, and will continue with thorough analysis of VDoc Specs parts giving them names so that it is easier to refer to particular notions throughout the chapter.

4.2.2 Virtual Documents by Example

We use a simple running example to give an initial intuition about VDoc Specs (Section 4.6 will tackle the real-world scenarios and justify VDocs in the TNTBase system). Let us consider the following document. We have a set of exercises with problem statements and solutions in our document collection. Say they are marked up in the following way:

Listing 4.2: An Exercise with Solution

```xml
<exercise>
  <problem>P</problem>
  <solution>S</solution>
</exercise>
```

Now we want to make them available in two forms: without solutions to students and with (master) solutions to the teaching assistants. In this situation, VDocs are well suited. To define a VDoc we provide a VDoc Spec. It consists of a skeleton and a list of external XQuery queries (we will use the same namespace prefix vx for VDoc Spec-specific elements as we used for a VXD):

Listing 4.3: A VDoc Spec for Practice Exercises

```xml
<vx:virtualdocument xmlns:vx="http://vxdoc.com/ns">
  <vx:skeleton xmlns:id="exercises">
      <dc:title>Exercises for GenCS</dc:title>
      <dc:creator>Michael Kohlhase</dc:creator>
      <omtext>The following individuals contributed to the document</omtext>
      <vx:xqinclude
        query="fn:distinct-values(vx:collection/*omdoc//dc:author)"/>
      <vx:return><omtext>\</omtext><vx:results/></vx:return>
    </omdoc>
  </vx:skeleton>
</vx:virtualdocument>
```

77
4. SCRIPTING XML DOCUMENTS

Conceptually, the skeleton consists of the top-level parts of the intended exercises document, where some document fragments have been replaced by embedded XQuery statements that generate them. In general, queries have the form:

Listing 4.4: An XQuery Fragment Reference

where \( q \) is an XQuery query and \( R \) consists of a result expression which may contain the \( \text{vx:result} \) elements that will be replaced with the results of \( q \). In line 7-9 we have such a query in a syntactic variant where \(<\text{vx:xqinclude query="q"}/R</\text{vx:xqinclude}>\) was used to abbreviate the primary query form above for queries that do not contain embedded elements. The result of this particular query would be a list of author names wrapped in \( \text{omtext} \) elements. The query in lines 15-18 uses another useful syntactic feature: it refers to the query by reference to enable reuse and sharing for VDoc Specs (see below). The result of this query would be a list of exercises of the form:

Note that the default and the \( \text{dc} \) namespace within the query are inherited from the dominating \( \text{omdoc} \) node.

Thus, the VDoc in Listing 4.3 indeed expands to the desired exercises document after XQuery processing. We can create multiple VDocs based on a single VDoc Spec. For instance, the above VDoc Spec’s XQuery statements can be changed so that they select problems in the directory where the VDoc resides. This is done by

\[\text{It is also possible to integrate a query from a separate module so that individual functions and variable declarations can be shared between queries.}\]
for $i$ in collection(./∗.omdoc)//exercise return $i/*[local-name() ne 'solution'], where the first "∗" in the query stands for the current directory. Thus we write a VDoc Spec once, and may use it for creating VDocs without solutions in different folders which are, presumably, distinguished by material topics.

The reference-based setup caters for a wide variety of reuse scenarios. For instance the document for the GenCS teaching assistants with solutions can be specified by the following VDoc Spec, which reuses the skeleton from Listing 4.3 and “overrides” the named query vx:SMLex:

```xml
<vx:virtualdocument xmlns:vx="http://vxd.com/ns">
  <vx:skeleton xml:id="exercises">
    <omdoc xmlns:dc="http://purl.org/dc/elements/1.1/">
      <dc:title>Exercises for Computer Science lectures</dc:title>
      <dc:creator>Michael Kohlhase</dc:creator>
      <omdoc>
        <dc:title>Acknowledgements</dc:title>
        <omtext>
          The following individuals have contributed material to this document:
        </omtext>
      </omdoc>
    </omdoc>
  </vx:skeleton>
  <vx:query name="xq.SMLex">
    for $i$ in vx:collection('/gencs/SML/∗.omdoc)//exercise return $i
  </vx:query>
</vx:virtualdocument>
```

Note that the new query does not remove the solutions. It is lexically captured by the named reference in the skeleton. The alluded approach of skeletons inclusion and overriding named XQuery statements may be compared to the method overriding in many object-oriented programming languages such as C++ or Java. It is also possible to have an empty vx:query element is the included specification. Then it becomes an abstract VDoc Spec, and can be compared, for instance, to abstract methods in Java or pure virtual functions in C++.

### 4.2.3 VDoc Specifications and Operational Semantics

In the previous section we briefly looked at the simple VDoc example, mostly at the VDoc Specifications that define VDocs. Informally speaking, a VDoc Spec is a document template with XQuery query inclusions and some other auxiliary elements that help to figure out how to populate query results within a resulting view. In Listing 4.5 we can see a slightly elaborated version of the VDoc Spec considered in the previous section. The VDoc Spec defines a VDoc that contains thematic lecture exercises together with their authors. It conforms to the RELAX NG Schema in Listing 4.1.

#### Listing 4.5: Example of a VDoc Spec

```xml
<vx:virtualdocument xmlns:vx="http://vxd.com/ns">
  <vx:skeleton xml:id="exercises">
    <omdoc xmlns:dc="http://purl.org/dc/elements/1.1/">
      <dc:title>Exercises for Computer Science lectures</dc:title>
      <dc:creator>Michael Kohlhase</dc:creator>
      <omdoc>
        <dc:title>Acknowledgements</dc:title>
        <omtext>
          The following individuals have contributed material to this document:
        </omtext>
      </omdoc>
    </omdoc>
  </vx:skeleton>
  <vx:query name="xq:collection('/exercises/∗.omdoc')//exercise">
    return $i
  </vx:query>
</vx:virtualdocument>
```
4. SCRIPTING XML DOCUMENTS

A VDoc Spec consists of a VDoc Skeleton (VDoc Skel), a number of named queries that are referenced from a VDoc Skel and arbitrary parameters that are used in the embedded queries. Let us consider these elements in order:

**VDoc Skeletons** contain a mixture between XML nodes and vx:xqinclude elements. The latter ones specify a single query and “the rules” how results of that query will be intermingled with other elements in a VDoc. The rules are enclosed into a single vx:return child element that, in turn, contains a mixture of any XML elements with zero or more empty vx:result elements.

To understand how VDoc content is produced let us consider the algorithm in XQuery pseudo-language (note the use of the XQuery processor function ξ) that takes a VDoc Spec XML element as an input and produces VDoc content of the document XDM type:

**Listing 4.6: Algorithm for expanding VDoc Specs**

```xquery
(: Processing the root of specification :) 
declare function vx:vdoc($vds as document−node()) { 
  let $skeleton := $vds//vx:skeleton
```
4.2 Virtual Document Fundamentals

if (empty($skeleton/*)) then
   (: vx:get−query function retrieves an associated query :)  
   ξ(vx:get−query($skeleton))
else
   vx:process($skeleton/*)
};

(: Processing the skeleton of the specification:)
declare function vx:process($node as node) {
   if (vx:is−vds−element($node, "xqinclude") ) then
      vx:process−vx−include($node)
   else
      if (not($node instance of element())) then
         $node
      else
         element {$node/name()} {
            for $child in $node/node() return
            vx:process($child)
         }
   }
};

(: Processing vx:xqinclude element: get a query and process its results :)
declare function vx:process−vx−include($el as element()) {
   let $query as xs:string = vx:get−query($el)
   for $result in ξ($query) return
   vx:process−result($el/vx:return, $result)
};

(: Recursively processing the vx:return element :)
declare function vx:process−result($node as element(), $result) {
   if (vx:is−vds−element($node, "result") ) then
      $result
   else
      if (not($node instance of element())) then
         $node
      else
         element {$node/name()} {
            for $child in $node/node() return
            vx:process−result($child, $result)
         }
   }
};

(: Actual call to retrieve VDoc contents :)
vx:vdoc($vds)

To summarize the algorithm verbally:

1. We take a vx:xqinclude element and obtain a query associated with it
2. We execute the query from step 1), obtain its results and iterate over them
4. SCRIPTING XML DOCUMENTS

3. For every result we get children of the vx:return element and substitute every vx:result element with a considered query result

4. We concatenate in order all children obtained from step 3)

5. The result of concatenation replaces the considered vx:xqinclude element

6. Repeat steps 1)-5) for all vx:xqinclude elements in a VDoc Spec

The algorithm is quite intuitive: let us look at the following example. The part of a VDoc:

```xml
<vx:xqinclude query="vx:collection('/exercises//.omdoc')//dc:creator/text()">
  <vx:return><omtext><vx:result/></omtext></vx:return>
</vx:xqinclude>
```

assuming for the sake of simplicity that results of the query

```xml
vx:collection('/exercises//.omdoc')//dc:creator/text()
```

are “Paul” and “John”, will be substituted by:

```xml
<omtext>Paul</omtext>
<omtext>John</omtext>
```

VDocs allow an arbitrary number of not nested vx:xqinclude elements in the VDoc Skel. A query can be defined in four ways: in the attribute, in the child element as text, as a reference to outside defined queries (described below) and as a reference to an external document (e.g. a file in the underlying VXD) which contains the query in question. It is worth mentioning that VDoc Specs may simply contain only references to skeletons in other VDoc Specs and differ just in queries or parameters. This approach becomes very handy when we want to leverage the same skeleton but tweak another parts of a VDoc Spec, i.e. queries or parameters (see below).

**Queries** Apart from being defined in vx:xqinclude elements, XQuery statement can also be described in separate vx:query elements, again as a text or as a reference to an external resource. In the former case a query must contain a name that serves as a link point from the VDoc Skel. There are no constraints on a query: it may reference, for instance, older revisions of documents (if an underlying system has a notion of different revisions like TNTBase does) or other VDocs. The “external” query definitions are beneficial when we want to override the queries for a particular VDoc Skel. Getting back to our example, assume that we want to embed only the statements of the assignments in our VDoc preserving the common structure. We do not need to modify a VDoc Skel. Instead we create a new VDoc Spec that references the existing one with an overriding XQuery statement.

```xml
<vx:virtualdocument xmlns:vx="http://vxd.com/ns">
  <vx:skeleton href="/basic−spec.xml" />
  <vx:query name="exercises.xq" />
</vx:virtualdocument>
```
Here we assume that the “parent” VDoc Skel can be found in the VDoc Spec under the path /basic-spec.xml. Thus we can “inherit” VDoc Skels recursively and override queries in any combination that comprised quite a flexible mechanism to reuse existing VDoc Skels and queries and override only parts when needed. Also it is possible to create VDoc Specs whose skeletons reference queries that are not present in the same VDoc Spec. This feature is comparable to e.g. Java abstract classes, i.e. such a VDoc Spec cannot be used as such, but can be referenced from another VDoc Specs that defines the missing queries. Again, these peculiarities are reflected in the RELAX NG schema in Listing 4.1.

Parameters Queries can reference variables that are not defined in the current context. Those can be externally defined in the vx:param elements outside the query. This approach separates computational logic of VDocs from the input data. Similarly to queries, parameters can also be overridden or be absent in any VDoc Spec. In the latter case, VDoc Specs that inherit the current VDoc Spec must define absent parameters. In our example in Listing 4.5, we are using two parameters: a list of topics for which we retrieve exercises ($topics) and a maximum number of returned exercises for each topic ($max). Such a mechanism considerably improves reusability and flexibility of VDocs. In our example parameters have type of a sequence of strings, however, an implementation of the VDoc concept is free to introduce its own XML syntax of VDoc Spec parameters to markup more complicated XDM types and implement an appropriate parsing.

4.2.4 VDocs as XML Database FS Entities

In Section 3.2.3 we have defined an FS for a VXD $X$. FSDs format left enough freedom for introducing another FS entities. Making use of it we define an FS entity for VDocs. Apart from having a unique name per directory a VDoc FS entity must have a link to the VDoc Spec that defines it.

Definition 4.4 (A VDoc FS Entity) A VDoc FS-entity (VDFSE) is a tuple \(\langle d, n, s \rangle\), where

1. \(d \in \text{xs:string}\) is a path of the parent directory.
2. \(n \in \mathbb{N}\) is a name of the VDoc FS entity.
3. \(s \in \text{xs:string}\) is a path of the corresponding VDoc Spec, where \(\xi(\text{vx:doc}(s))\) is defined and conforms to the VDoc Spec schema.
We will call a VDoc FS entity a VDoc as well when we need a formal definition of a VDoc.

Given the definition above it is easy to represent a VDoc FS entity in FSDs:

document {
    <vx:fsd path="d"/>
    ...
    <vx:vdoc name="n" vds="s"/>
    ...
}</vx:fsd>

Note the additional attribute vds that points to the corresponding VDoc Spec.

An FSD example containing a VDoc (i.e. a VDoc FS entity) is:

<vx:fsd path="/exercises"/>
<vx:dir name="graphs"/>
<vx:dir name="prolog"/>
...
<vx:file name="exercises-studs.vds.xml"/>
<vx:file name="exercises-TAs.vds.xml"/>
<vx:vdoc name="exercises-studs.vdoc" vds="/exercises/exercises-studs.vds.xml"/>
<vx:vdoc name="exercises-TAs.vdoc" vds="/exercises/exercises-TAs.vds.xml"/>
</vx:fsd>

In the example above there is the exercises folder with subfolders for assignments based on their topic, VDoc Specs for VDocs with solutions and without, as well as VDoc FS entities pointing to the corresponding VDoc Specs.

Since we now have a special FS entity for VDocs we can redefine the vx:doc function (see Theorem 3.1) so that it works both for physical documents and VDocs likewise.

Listing 4.7: The vx:doc function extended to VDocs

declare function vx:doc($path as xs:string?) as document-node()? {
    if (empty($path)) then
        () (: an empty sequence if $path is empty :)
    else
        let $dir-path := fn:substring-before-last($path, "/")
        let $name := fn:substring-after-last($path, "/")
        let $fs-entity = vx:fsd($dir-path)/vx:*[@name = name]
        if (fn:empty($fs-entity) then
            fn:concat('No document under path ', $path))
        else
            let $type = $fs-entity/local-name(.)
            if ($type eq "file") then
                fn:collection([ns:metadata('vx:path', .) = $path]
            else if ($type eq "vdoc") then
                vx:vdoc(vx:doc($fs-entity/@vds))
    };

It is easy to see that the updated version of vx:doc behaves absolutely the same for physical documents and therefore an underlying FS-aware database remains so. But now the same XQuery function is able to return VDocs and therefore use them in more sophisticated queries or even in queries of other VDoc Specs.

**Example 4.4 (Querying a VDoc)** Following our exercise example, assume that we want to get a number of authors that have contributed to the exercises in a VDoc. The following query does what is intended:

```
fn:count(vx:doc(’/exercises/exercises−for−students.vdoc’)//dc:author)
```

VDoc FS entities encoded into FSDs leave enough freedom to (re-)define parameters in a VDoc Spec. Since parameters are encoded as XML in VDoc Specs, they also can be encoded as a child element of the VDoc FS entity element. In this cases parameters defined in a VDFSE take priority over those defined in a VDoc Spec. Of course, the vx:vdoc function must be aware of that and process parameters following the precedence rule.

Before advancing to the next section let us give a definition of VDoc content:

**Definition 4.5 (VDoc Content)** VDoc content of a VDoc under path \( p \) is an XDM instance defined as \( \xi(vx:doc(p)) \).

### 4.2.5 VDocs as XML Database Views

**Lemma 4.2** For every VDoc Spec document \( sd \) from \( X_D \) there exists \( q \in XQ \) such that \( \xi(vx:vdoc(sd)) = \xi(q) \)

**Proof:** Look at Listing 4.6. The XQuery pseudo-code is actually well-formed XQuery apart from the not mentioned \( vx:get−query \) function and applications of the XQuery processor function \( \xi \). Since XQuery queries input and output are closed with respect to XDM, the values of \( \xi \) in the algorithm code belong to XDM. Since every instance of XDM can be interpreted as a sequence of zero or more elements, the values of \( \xi \) are applicable to the \( for \) loop, and therefore the whole expanded code is an instance of the XQuery language. Thus we found \( q \) which is equal to this instance.

Lemma 4.6 allows us to introduce the following definition:

**Definition 4.6 (VDoc Spec Expansion)** We call a VDoc Spec expansion a function \( exp : X \rightarrow XQ \), that for every VDoc Spec \( s \) and an XML database \( X \) produces an instance of XQuery language as defined in Lemma 4.2.

**Theorem 4.1** Let \( \langle d, n, s \rangle \) be a VDoc, then \( \nu = \langle q, p \rangle \) is an XML database view, where \( q = exp(vx:doc(s)) \) and \( p = fn:concat(d, ’/’, n) \).
4. SCRIPTING XML DOCUMENTS

Proof: By definition of the XML database view the following must hold \( \xi(vx:doc(p)) = \xi(q) \) which is equivalent to \( \xi(vx:doc(fn:concat(d, '/', n))) = \xi(exp(vx:doc(s))) \). That can be rewritten using Definition 4.6 as \( \xi(vx:doc(fn:concat(d, '/', n))) = \xi(vx:vdoc(sd)) \), where sd is a VDoc Spec for the VDoc. The document node for the VDoc Spec is located under FS path s, and according to the definition of the function vx:doc on Listing 4.7 the VDoc content will be obtained via \( \xi(vx:vdoc(vx:doc(s))) \). Since \( p = fn:concat(d, '/', n) \) from Listing 4.7 we can deduct that \( sd = \xi(vx:doc(s)) \), which proves the theorem. □

The theorem above allows us to define XML database views via VDoc Specs, and treat VDocs as true XML database views. Given Theorem 4.1 we know that VDocs indeed conform to the general notion of the database views defined in Definition 4.1. Although we have an easier syntax to define XML database views, we do not lose expressiveness of the XQuery language since VDoc Specs may also consist of a single XQuery expression. But the richer syntax for defining XML database views allows us to make them editable which is the subject of the next section.

4.2.6 Editing Virtual Documents

Database views can be editable. That means that the edited parts are transparently propagated to its original physical locations in the database, if possible. In this section we will show that by defining XML database views via VDoc Specs, it is possible to make VDocs editable as well when changed fragments of VDocs that were injected from some physical XML documents in the XML database get propagated back to them. No efforts are required from the user side to track the origins of changes. In order to understand how XML views can be edited we have to work out a formalism. The core idea consists of annotating XML fragments (with source references) in the VDoc content that actually were retrieved from some physical XML documents. We start with introducing an XPath address of XML elements.

4.2.6.1 Editing Fundamentals

Definition 4.7 (An XPath Address) Let \( X_E \) be a set of XDM element nodes and \( e \) is from \( X_E \) with a root document node \( r \in X_D \), then an XPath expression \( a \) of the form \( /xs:QName[i_1]/.../xs:QName[i_n] \) is called an XPath address of the node \( e \) in the document \( r \) iff \( \xi(r, a) = e \), where \( i_k \) are integers bigger than 0.

Since XPath is a subset of XQuery we can apply an XQuery processor function to any XPath expression. Document \( r \) is supplied as a context node to the XQuery processor.

Example 4.5 (An XPath Address) Consider the following XML document:

```xml
<persons>
  <person type="Doctor">
    <first-name>John</first-name>
    <last-name>Smith</last-name>
  </person>
</persons>
```
4.2 Virtual Document Fundamentals

The XPath expression `/persons[1]/person[1]/country[1]` addresses the XML element `<country>Germany</country>` and is indeed an XPath address. Whereas the XPath expression `(persons/*/country)[1]` “selects” the same node, it is not an XPath address since it is not of the form `/xs:QName[i]/*/xs:QName[in]` missing positional predicates for every step on the child axis. The following XPath address selects the `<country>Sweden</country>` element: `/persons[1]/person[2]/country[2].`

Example 4.6 (An XPath Address Construction) The following XQuery function constructs an XPath address from the element node:

```
declare function vx:xpath-address($el as element()) as xs:string {
fn:string-join(
  for $i in $el/ancestor-or-self::* return
  let $cnt := fn:count($i/preceding-sibling::*[fn:node-name(.) = fn:node-name($i)]) + 1
  return fn:concat(name($i), ", [", string($cnt), "]")
), "/"
};
```

Lemma 4.3 An XPath address uniquely defines a node within an XML document.

Proof: Since an XML document is a tree, there is only one path to every node from the root. An XPath address on each step uniquely selects the next node in that path due to its positional predicate. Since the choice of the child node on each step is unambiguous, the whole path generated by an XPath address is unique.

Definition 4.8 (A Node Origin) A node origin of an XDM item in XML is a sequence defined by the following XQuery function:

```
declare function vx:node-origin($item) {
  if ($item instance of element()) then
    let $uri := $item/document-uri(.) return
    if (empty($uri)) then
     ()
    else
      (fn:doc($uri)/ns:metadata("vx:path", .), vx:xpath-address($item))
  else
    ()
};
```
A node origin is an empty sequence for all XDM data types except element nodes. When it is non-empty it contains a path of the document within XD to which this element belongs and its XPath address.

**Example 4.7 (Node origins)** Assume we have the following document under the path /persons/catalog1.xml:

```
<persons>
  <person type="Doctor">
    <first-name>John</first-name>
    <last-name>Smith</last-name>
  </person>
  <country>Germany</country>
</persons>
```

The node origin of the element `<country>Germany</country>` is `("/persons/catalog1/xml", "/persons[1]/person[1]/country[1] ")`, whereas the node origin of the attribute type is an empty sequence since it is not an element node. The result of the XQuery expression

```xquery
vx:node-origin(document{<a><b/></a>}/a[1]/b[1])
```

is also an empty sequence since the document in the query is a constructed node and therefore has no document URI.

**Lemma 4.4** A node origin uniquely identifies an XML element in XD.

**Proof:** Since an FS path uniquely identifies a document within XD, and a XPath address uniquely identifies an element within a document, then a pair of those (i.e. a node origin) uniquely identifies an element in the database. □

### 4.2.6.2 Annotating XML Elements

We can utilize the knowledge that node origins uniquely identify XML elements in the database and annotate those XML elements in the VDoc content that actually came from physical XML documents in the database. For this we have to slightly modify our algorithm in Listing 4.6. We will use the idea of XQuery Scripting Extensions [Sne+10] to introduce a side effect to the processing of the `<tnt:result/>` elements:

```xquery
declare function vx:process-result($node as element(), $result) {
  if (vx:is-vds-element($node, "result")) then
    (: Annotating with a node origin :) ...
```

88
4.2 Virtual Document Fundamentals

vx:annotate($result, vx:node-origin($result));

else

  if (not($node instance of element())) then
    $node
  else
    element {$node/name()} {
      for $child in $node/node() return
        vx:process-result($child, $result)
    }

};

For now we assume that the vx:annotate function somehow preserves the node origin of the element, and Section 4.3.1 will look at the possible implementations of that function. We will also assume that the function vx:get-node-origin($node) returns node origins of those XML nodes in the VDoc content that has been annotated upon VDoc Spec processing.

**Definition 4.9 (Editable and Immutable Nodes)** Let a node $node be any node from some VDoc content. This node is called **editable** iff the sequence $\xi(\node/ancestor-or-self::*//vx:get-node-origin(.))$ is non-empty. Otherwise node $node$ is called **immutable**.

The intuition behind the definition above is that in the VDoc content only those nodes are editable that have a node origin or a part of a node that has a node origin. In other words, only those XML fragments are editable that are a part of some physical documents in the underlying database.

Let us get back to the exercise example in Section 4.2.2. Consider that we annotate XML elements with node origins by means of XML attributes. A part of an annotated VDoc content can look like:

```xml
<omdoc xmlns=...>
  <exercise>
    <problem vx:path="/gencs/SML/prob1.omdoc"
              vx:xpath-address="omdoc[1]/exercise[3]/problem[1]">P</problem>
  </exercise>
</omdoc>
```

Since this particular exercise is contained in the XML document under the path /gencs/SML/prob1.omdoc we can extract node origins for the exercise XML element. The vx:xpath-address attribute tells us that the retrieved element is the first problem of the third exercise in the document mentioned above. Since this element contains node origins it becomes editable, and we theoretically know where we should propagate changes to. It is worth noting that using attributes is a very simple and error-prone way to encode node origins. Other approaches are discussed in Section 4.3.1.
4. SCRIPTING XML DOCUMENTS

4.2.6.3 Persisting VDoc Changes

There is one more step left to conclude the theoretical part of VDoc editing: propagating changes back to the database. To keep VDoc editing reasonably simple we require that the VDoc content remains the same between its retrieving and submitting changes back. In order to guarantee this property in the general case we require that the content of the database has not been changed in the meanwhile. That can be assured by the optimistic concurrency control method well-known in the database field [HT81].

Remember from the previous section that we allow editing only editable nodes of the VDoc content. Here is an algorithm for propagating changes back for the modified VDoc content $v$ under path $p$ in XQuery pseudo-language:

```
declare function vx:get-editable-root($node as node()) as xs:boolean {
  $node/ancestor-or-self::*[vx:get-node-origin(.)]
}

declare updating function vx:submit-changes($vdoc-path as xs:string, $modified-vdoc) {
  let $original-vdoc := vx:doc($vdoc-path)
  return (: Check if the VDoc has been changed in the meanwhile :) 
  if (not(vx:same-version($original-vdoc, $modified-vdoc))) then 
    fn:error(QName('http://vxd.com/ns', 'Conflict'), 'DB has been changed') 
  else 
    for $node in vx:diff($original-vdoc, $modified-vdoc) return 
    let $editable-root := vx:get-editable-root($node) return 
    if (empty($editable-root)) then 
      fn:error(QName('http://vxd.com/ns', 'Conflict'), 'An immutable mode was changed') 
    else 
      let $origin := vx:get-node-origin($editable-root)
      let $path := $origin[1]
      let $xpath-address := $origin[2]
      replace node $node with $modified-vdoc($path), $xpath-address
  }

vx:submit-changes(p, v)
```

We observe that:

- The replace XQuery Update statement requires a single node at its target. This is guaranteed by Lemma 4.4.

- We utilize the function vx:same-version as a part of the optimistic concurrency control method.

- The function vx:diff returns parts of an XML tree that were changed in the XML document. Regardless its implementation the differencing algorithm can be used as it is. An elaborated survey of XML-diff algorithms and tools has been made in [Pet05].
• In the algorithm we use an XQuery processor function to evaluate an XPath expression in a context document node. Some of XQuery processors such as Saxon provide dynamic evaluation of XPath. As an alternative, pure XQuery implementation can be used for that purpose. External XQuery functions do the trick as well.

4.2.6.4 The Editing Workflow

Figure 4.1: A VDoc workflow

Figure 4.1 exemplifies three phases of VDoc editing:

P1 A user retrieves VDoc content.

P2 A user modifies VDoc content.

P3 A user submits the modified VDoc back to the underlying system.

4.2.6.5 Editing Limitations and Solutions

We have described the basic approach for VDoc editing. Despite its transparency and convenience it has a few limitations:
4. SCRIPTING XML DOCUMENTS

- Requiring that the database content is not changed while editing VDocs might be too restrictive. That can be relaxed by annotating editable elements with its own version when it was changed last time. Then we just need to require that the versions of the changed editable elements are the same. It is another variant of the optimistic control methods. Having versioning mechanisms like in VXDAs allows us to easily track those versions by reading the change log. However, we consider this problem as an optimisation task which does not influence the validity of the approach. We recommend to implement more relaxed locking for those use cases when the content of the database is changed frequently.

- Designers of VDoc Specs for VDocs should note that the number of editable nodes may be different for semantically equivalent specifications but with different balance between static VDoc Spec elements and included XQuery expressions. Consider the following VDoc Specs:

```xml
<vx:virtualdocument>
  <vx:skeleton>
    <omdoc xmlns=...>
      ...
      <vx:xqinclude>
        <vx:query>vx:doc(...)//problem</vx:query>
        <vx:return><exercise><vx:result/></exercise><vx:return/>
        </vx:query>
      </vx:xqinclude>
      ...
    </vx:skeleton>
  </vx:skeleton>
</vx:virtualdocument>
```

versus

```xml
<vx:virtualdocument>
  <vx:skeleton>
    <omdoc xmlns=...>
      ...
      <vx:xqinclude>
        <vx:query>
          for $p in vx:doc(...)//problem return $p
        </vx:query>
        <vx:return><vx:result/><vx:return/>
        </vx:query>
      </vx:xqinclude>
      ...
    </vx:skeleton>
  </vx:skeleton>
</vx:virtualdocument>
```

In the first form the logic of wrapping problems into the exercise element is encoded in the XML elements of the VDoc Spec itself, whereas in the latter example this logic is embedded into the XQuery expressions. Although the VDoc content will
be identical in both cases, the latter example makes problem elements immutable since the query of the VDoc Spec returns constructed XML elements that do not have document URIs. To overcome this problem even in the latter case a developer of XQuery expressions can manually execute the vx:annotate function. So the modified XQuery query will be:

```
for $p in vx:doc(...)//problem
  <exercise>
  {
    vx:annotate($p, vx:node-origin($p));
    $p
  }
  </exercise>
```

However, this method requires additional coordination from the developer side, and therefore it is recommended to avoid it if possible.

- When a node with the same origin is presented more than once in the VDoc content, then according to the algorithm only one modification will be persisted in the database. Although it is rather uncommon to have such a situation, it is still possible. An algorithm implementation may decide to forbid such changes completely.

### 4.3 Integrating with an XML Database

Without loss of generality we will assume that we are going to embed a virtual document concept into a versioned XML database presented in the previous chapter. But as mentioned before we do not require a versioning aspect for VDocs.

From the engineering perspective it is beneficial to implement the VDoc concept in pure XQuery as much as possible so that the implementation is reusable among XQuery processors with minimum adaptations.

The following aspects can be hard, if possible at all, to implements in pure XQuery:

- In our algorithm we used references to the XQuery processor function $\xi$ that implies dynamic evaluation of queries from other XQuery statements. Whereas simple XPath evaluation is easily possible in XQuery, an arbitrary XQuery expression is hard to evaluate in the XQuery query unless we have an XQuery compiler written in XQuery. An alternative would be to compile a VDoc Spec into the XQuery query using your own XQuery interpreter. The problem of preserving node origins arise.

- Depending on the way we preserve node origins it may be hard to accomplish it without XQuery scripting extensions, or impossible to do so without external XQuery functions.

- Implementing XML-diff in XQuery requires some work. However, it has been implemented by the author and is available in [Zho+11].
4. SCRIPTING XML DOCUMENTS

We do not expand further on this topic since implementation decisions are highly dependent on the use cases and the technology stack being used.

4.3.1 Embedding Node Origins

We have foreseen several ways to embed node origins:

- **XML attributes** This approach was already exemplified in Section 4.2.6.2. The drawback of this approach is that XML attributes are intermixed with a content of a VDoc and thus they can be edited by a user or forbidden by an associated schema. The former problem can be solved by forbidding such changes upon submission. The latter issue can be resolved by adjusting the corresponding schema.

- **XML comments** We put them before the editable element. This approach is semantically equivalent to the approach with attributes. The positive aspect is that an auxiliary information is not intermixed with the editable elements and does not affect the validation phase. Also we have the same problem of deleting those comments as in case with XML attributes.

- **Processing instructions** The semantics is the same as in the two methods described above, however, of a different form. Processing instructions are associated with the editable fragments and must not be edited in order to be able to submit changes.

- **External data structure** It is possible to leave VDoc content “clean” and store the node origin information in an supplementary data structure (like SVN stores modifications in the file system side by side with files located in the working copy). The advantage of this approach is that a user cannot accidentally change the node origin information. However, this approach requires preservation of the state during editing and usually involves greater complexity on the client side.

Picking the appropriate method is a balance between the level of desired complexity, a shape of VDoc content and simplicity for an end-user.

4.3.2 Extending the VXD Toolkit

The VDoc concept naturally fits into the VXD toolkit introduced in Section 3.3. By now we have discussed all relevant theoretical concepts and prepared to extend the VXD toolkit. No additional components are required – just the modification of the existing ones, namely an XQL and xAccessor. Figure 4.2 shows the VXD toolkit architecture extended with the VDoc concept.

The idea is to extend an XQL with the implementations of VDoc-related XQuery functions discussed in this chapter. Some of these functions may be external, and therefore xAccessor must be extended as well to hold target language implementations of the external XQuery functions. Additionally, xAccessor must bridge a VXD interface
4.3 Integrating with an XML Database

![Diagram of VXD Toolkit and VDocs](image)

Figure 4.2: A VXD Toolkit and VDocs

and calls to the VDocs XQuery module. A subcomponent of xAccessor in charge of these tasks is called a Virtual Document Processor (VDP). A VDP is responsible for:

- Maintaining VDoc FS entities.
- Redefining XQuery parameters in VDFSEs used in the corresponding VDoc Spec.
- Supporting editing workflows (cf. Section 4.2.6.4). VcsAccessor helps us with preserving revision history of changed documents.
- Create versioned snapshots and materializing VDocs (See Section 4.3.3)

The last item has not been considered so far, and it is a subject of the next section.

4.3.3 Versioned Snapshots and Materializing VDocs

In the database theory materialized views can mean slightly different concepts:

1. **Snapshots** of a view taken at a certain point in time.

2. The view content is periodically updated and cached (materialized) in the database to enable more efficient access. Depending on access patterns the view re-computation may be done less frequently at the cost of data being slightly out-of-date. This approach works well for the scenarios where view computations take significant amount of time while the content of an underlying system is changed rarely. The more complex problem is to recompute a materialized version as needed keeping it always up-to-date, and at the same time to perform as few recomputations as possible. In this work we only acknowledge the problem and the author did not attempt to solve it.

The items above are covered in the next two sections.
4. SCRIPTING XML DOCUMENTS

4.3.3.1 VDoc Snapshots

To understand the motivation behind having snapshots let us get back to our exercise example. Assume that we modified our XQuery expression so that it generates exercises sheets only from the newest problems whereas new ones are constantly committed to the VXD. Periodically we want to generate exercises sheets and persist them in the database so that they can be used later by younger generations of students. That is where VDoc snapshots prove to be useful. If we are interested in only one snapshot at a time, we might persist it under the same path, and a VXD will preserve the change history allowing to track evolution of exercise sheets over time.

4.3.3.2 VDoc Materializing

While the author did not conduct considerable research on the VDocs materializing, we could not ignore this topic and therefore cover a possible naive realization of materializing in the scope of the VXD toolkit.

We envision two simple ways of dealing with materialization.

- Cache VDoc content upon retrieval and track a HEAD revision number when the VDoc computation occurred. Invalidate cache when next retrieval occurs and a HEAD revision is different from the tracked one. In this case we can guarantee an up-to-date VDoc content.

- Once a VDFSE is created, a dedicated background process computes VDoc content and caches it. Re-computation is done periodically, and therefore we can not ensure that VDoc content is always up-to-date. However, in some use cases it may be an acceptable trade-off between consistency and efficiency.

In both cases the question arises: how do we figure out whether a cached version of a VDoc is outdated? There is no common way to predict it, for instance, if a VDoc Spec contains a query that returns current time and date, then the cached content will always contain obsolete time. Therefore in some cases caching would not be of any help. A VDP should provide to the user an interface for indicating which VDocs can be an object of meaningful caching. Some default value may be set by a VDP automatically.

4.3.4 VDocs and Versioning

VDocs can take advantage of the versioning model of the VXD. The following scenarios make the VDoc concept even more powerful in the scope of the VXD toolkit.

- VDFSEs can reference not only the HEAD version of a certain VDoc Spec, but link to a particular revision of it. This allows a VDoc Spec developer to have a "stable" VDoc that is linked to a fixed well-developed version of the VDoc Spec, whereas a new version of the VDoc Spec that is still in progress would define a developer version of a VDoc.
4.4 Related Work

In this section we briefly discuss some of the relevant related work items that the author has explored.

4.4.1 Drupal

Drupal [Dru], an open-source content management system, has a notion of XML views. Exploring this concept showed that Drupal XML views are quite different from the VDocs concept. The major differences are:

- Drupal views operate on a single XML file.
- Drupal uses only XPath to select nodes.
- Most importantly, Drupal focuses on showing parts of an XML file in a human-readable presentation such as a table or a list.

Given the items above, a Drupal approach is quite different from VDocs. Some other related systems such as Wordpress [Wor] have a similar concept of views.

4.4.2 Materialized XML Views

Interesting work has been performed in [ES05; Nil06] about materialized XML views that are defined by an XQuery statement. The work focuses on theoretical aspects and improving efficiency of the views by advanced materializing and keeping a materialized version consistent. In this respect our work on VDocs differs from the mentioned one in a way that we prioritize the elaboration of a more powerful and easy-to-use concept, integrating it at the same time with a versioning model. Performance of VDocs has not been evaluated carefully at the moment. Unlike the mentioned work, in order to make views always up-to-date we suggest recomputing them on every request, and their materialized versions serve as snapshots in a certain point of time.

The following points make the VDocs concept more advanced:

- VDoc Spec queries can refer to a particular revision of documents in a repository (cf. Section 3.3.2.2). Moreover, a revision can be presented as a parameter in a VDoc Spec, therefore one VDoc Spec can serve as a view for a variable revision. In our exercise example we may want to change the referenced XQuery expression that selects all exercises and authors for a particular parametrizable revision number. In this case we could see which exercises are present in a certain revision and which exercises were added/deleted over time using XML-diff techniques. Note that editing of elements from previous revisions must be forbidden since revision history is immutable in our versioning model. The pragmatic solution would be that the vx:annotate function ignores such nodes.
4. SCRIPTING XML DOCUMENTS

- The mentioned work uses only subset of XQuery: XPath expressions, FLOWR statements and element constructors. Furthermore, XPath expressions can only use two XPath axes: child and descendant. These constraints naturally limit the expressiveness of XML views, unlike the case of VDocs where the full power of XQuery can be utilized.

- A view is defined as a named XQuery query which is as powerful but less beneficial than defining VDocs using VDoc Specs (Section 4.2.3).

- The mentioned work does not focus on implications of the views inside an XML database, especially how it affects a versioning aspect if present.

To conclude, the latter related work item presents a valuable contribution to the XML workflows regarding efficiency sacrificing the area of XML views application, whereas the VDocs approach is focused on expressivity, flexibility and integration with versioned XML databases. More work for the VDoc concept regarding automated materialization needs to be done.

4.5 Implementation in TNTBase

The TNTBase system provides an implementation of VDP. Here are the relevant design decisions (for technical details see [Zho+10]):

(1) A VDP is realized as an interpreter, that is VDP reads a VDoc Spec together with parameters and produces VDoc contents without compiling it into an intermediate XQuery expression.

(2) The REST interface of TNTBase has been extended to expose VDocs functionality. Parameters to VDoc queries can be provided dynamically in the URL.

(3) All VDoc functionality mentioned in this chapter is also accessible to an end-user via XQuery functions (mostly external).

(4) VDocs are the first class citizens in the TNTBase file system. VDFSE parameters, a link to a VDoc Spec and its revision are stored in the corresponding FSD (refer to Section 3.2.3). The tnt:doc function is used to access VDocs as well.

(5) A VDP is implemented in XQuery with usage of several external functions written in Java. External functions are used, for instance, for dynamic query evaluation from another XQuery expression, namely for a VDoc Spec expansion getting the revision information from a repository (to perform optimistic locking) and committing changes under a certain path (to propagate changes to original files once an edited VDoc has been submitted back).

---

1. In TNTBase it is possible to create a VDoc based on former revisions of VDoc Specs
2. To date, DB XML does not provide a mechanism for dynamic query evaluation
(6) Parameters for VDocs can be presented in a VDoc Spec itself, associated with a VDFSE or provided dynamically upon VDoc content evaluation. The latter options get higher priority over the former ones. Parameters are limited to a sequence of strings (xs:string*), which was sufficient for our use cases. However, it would be straightforward to extend the list of supported parameter types.

(7) To query VDocs and test VDoc Specs, the following XQuery functions are provided:

- `tnt:vdoc($path as xs:string?, $params as element()?)` - retrieving VDoc content based on the dynamic parameters provided as an argument in a wrapped XML element.
- The functions `tnt:vdoc-spec($spec-path as xs:string?)` and `tnt:vdoc-spec($spec-path as xs:string?, $params as element()?)` are similar to the one above with an exception that they retrieve content based on a VDoc Spec only without a need to have a VDFSE for that spec. It is useful for debugging purposes.

(8) There is no VDocs caching support in TNTBase yet because the frequent change of the document collections used in our experiments makes caching unimportant.

(9) After VDoc editing the VDP utilizes the author’s XQuery XML-diff implementation [Zho+11] that also classifies the elements that differ to editable and immutable.

(10) There is an implementation of `vx:annotate` function in XQuery that utilizes an XQuery function for computing node origins. Embedding of node origins is done in the simplest way via XML attributes.

The VDoc concept and its VDP implementation in TNTBase proved to be useful as shown in the next section.

### 4.6 VDoc Use Cases

In this section we discuss several real-world use cases of the VDoc technology. We experimented with those use cases in the realization of VDocs in TNTBase. The discussion here is complemented with a TNTBase sandbox installation [Zho] that supplies a RELAX NG schema for VDoc Specs and shows VDoc queries and VDoc Specs of our use cases in action.

#### 4.6.1 Automated Exam Generation

This is a dogfood use case from our academic practice, and is (partially) used in day-to-day operation: **Michael Kohlhase** teaches a first-year, two-semester introduction to Computer Science at Jacobs University and – over the last nine years – has accumulated

---

1 In the source code the elements that differ are classified to non-critical and critical, correspondingly
4. SCRIPTING XML DOCUMENTS

a collection of about 2000 homework, quiz, and exam problems encoded into the OM-
Doc format [Koh06b]. For the courses we need to prepare regular four exams, four
“grand tutorial test exams” and two make-up exams per year. While the homework
problems are typically new, we can assemble the exams from it semi-automatically with
a VDoc Spec that generates random exam sheet based on the input list of topics we
intend to cover throughout an exam.

There are two kind of proper exams: midterms and finals. Midterms usually are
meant to be for 1 hour, although sometimes it takes up to 75 minutes, whereas finals
are designed for 2 hours. Thus we also provide an exam duration as an input parameter
for our exam VDoc. Changing only this parameter together with the topic list allows
us to get different exam sheets that do not exceed the certain time and cover desired
topics. All necessary information is encoded into the problems as RDFa metadata
annotations [LK09]. Our XQuery for a VDoc Spec takes care about adjusting the timing
closely to the provided limit. When VDoc content is generated, it can be rendered
by utilizing XSLTs and the JOMDoc library [Jom] developed in our research group.
Everything is embedded into TNTBase, and once an exam VDFSE is created it is a
matter of one click to get a unique human-readable exam sheet for students.

VDoc editing facilities also find an application in our use case. Before giving gen-
erated exam to students we test it on our teaching assistants that may express some
suggestions how to improve particular problems. Then we edit the contents of an exam
VDoc and commit it back – all modifications are automatically patched into original
XML sources. If one does not like a particular problem to be included into exam, we
can adjust a VDoc parameter that excludes it from the exam.

The most important advantage of the exam generation approach is that we write a
VDoc Spec once and reuse it next semesters by simply adjusting few VDoc parameters.
When one is satisfied with the exam presented, it can be materialized and saved in the
repository as a file which can be referenced in the future to keep track how students
performed on different assignment and figure out what their weaknesses are.

Although the presented approach already meets our requirements, some issues can
be addressed. For instance, we might want to take total exam difficulty into account
to generate exams that do not exceed a certain duration and that have difficulty in a
certain range (again difficulty information is embedded into problems XML). This will
lead to a more complicated queries for a VDoc Spec. Apart from generating exams,
this use case might be used by students that are willing to sharpen their knowledge:
they could generate practice sheets starting from easy tasks and end up with the com-
plex ones. Some parameters in e.g. cookies may keep track of what exercises already
appeared in the practice sheet, and a VDoc will never show them again. It could easily
be done by providing dynamic parameters to a VDP upon VDoc contents retrieval.

4.6.2 Multiple Versions of Documents

In most scenarios with long-lived documents, we encounter the problem of document
versions. Let us consider the case of W3C specifications such as XQuery 1.0/1.1, XPath
1/2, XML 1.0/1.1., or even MathML 1.0/1.0.1/2.0/2.0(2e)/3.0. They are encoded in
the XML format XMLSpec \[\text{Xmla}\], so TNTBase and VDocs apply. Usually some parts of specification remained the same, while other parts change between the versions, and it is an important task to track the differences. For this use case we are experimenting with XML 1.0 and 1.1 specifications to supply the user with a view that will show only the relevant changes in the formal parts of specification branches. It is rather simple to provide a \textit{Diff VDoc} via an XQuery that summarize changes in formal parts (the rules of the XML grammar are marked up by special elements in XMLSpec), ignores document order (grammars are sets, not lists of rules), and presents them as XMLSpec documents upgraded with difference alternatives. This VDoc gives a user better understanding in which direction the development is going and what changes are intended ones and which are made by mistake. Our \textit{Diff VDoc} is also editable that allows a user to fix obvious bugs right on spot, without navigating to the source files. Once the Diff VDoc is created in TNTBase it can be reused to filter only relevant differences as well as transparently editing them, all in one place. Currently W3C stores specifications in a CVS repository, but does not make use of its differencing facilities for version tracking as diff is text-based and outputs even less and least relevant differences. Note that the Diff VDoc encapsulates a particular notion of relevance in the filtering part, which may need to be explained in a document preamble. Thus the representational form of a VDoc which mixes document parts and queries is beneficial. Moreover, there can be multiple Diff VDocs for tracking (and editing) various aspects of the differences in the specifications. Such Diff VDocs may even take over the role of conflict editors we currently have in VCS-aware integrated development environments.

4.6.3 Managing Document Collections

The exam generation use case described above can be seen as a special case of managing (here extracting custom documents from) a collection of primary (\textit{content}) documents and creating secondary documents from them that aggregate parts of the content. These secondary documents can either be used for communication to the outside (\textit{payload} documents) or for management of the document collections. In this terminology, the exams above can be seen as the payload documents derived from the content documents in the problem collection. That is where VDocs may naturally come into play as we have seen above.

A very simple application of VDocs in payload documents are queries for a table of contents (collecting all sectioning elements in a narrative document), the references (collecting all citations, sorting them, and completing them with information from a bibliographic database), or an index. In DocBook \[\text{WM08}\] these aggregated document parts generated by XSLT stylesheets in the presentation phase, which may incur performance bottlenecks in practice, since this is not supported by indexing and caching. Moreover, VDocs conceptually separate the issue of auto-aggregation and presentation, which allows to support workflows such as previews/editing of aggregated document parts and materialization (e.g. of branches and tags) for archiving.

Another simple application of VDocs in technical payload documents is in XML-based literate programming \[\text{Knu92}\], where program text is intermingled with its doc-
4. SCRIPTING XML DOCUMENTS

ocumentation and explanation in a single document. Here a VDoc can be used to extract the program text (with comments that cross-link to) from the literate source. As a concrete XML-based example take the XMLSpec-based source of the MathML 3 Recommendation from which we generate the MathML 3 RELAX NG Schema. A VDoc would have considerably simplified this process.

We have already seen Diff VDocs as examples of management VDocs for version management in the last section. But VDocs can also support proofreading, a very important task in the document life cycle. Often one wants to proofread special aspects of a document, e.g. whether certain technical terms are used consistently. For this we can quickly specify these terms as parameter to an XQuery that assembles all paragraphs that contain them. Then we can proofread (and edit) the text passages, commit them back to the collection, and advance to other proofreading tasks.

4.6.4 Refactoring Ontologies

We are using VDocs for experimenting with refactoring of OWL Ontologies that are written in XML syntax. We have chosen the OWL 2 XML syntax, the straightforward XML encoding of the functional-style syntax of OWL 2, in terms of which the direct semantics is specified.

Refactoring in software engineering is commonly defined as “a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior” [Fow]; this definition is also valid for ontologies. Typical ontology refactorings include splitting an ontology into several modules, or, conversely, merging multiple ontologies, moving axioms to another module of the same ontology, or rewriting axioms to semantically equivalent but shorter or longer forms (e.g., in description logics, rewriting the two axioms \( A \sqsubseteq B, A \sqsubseteq C \) into \( A \sqsubseteq B \sqcap C \)).

To help ontology developers not to break things, we provide a two-phase editing workflow. In the first phase a developer can preview ontologies changes using a VDoc which utilizes XQuery transform functions from XQuery Update Specification to perform refactorings. In the second phase, when a developer is satisfied with results he can materialize a VDoc thus obtaining a refactored ontology as a usual document in a repository. Of course, changes should be carefully tested before advancing to the second phase. The VDoc Specs of refactoring VDocs are maintained in the same repository as the ontologies and can collaboratively be refined by the ontology developers.

For more detailed information concerning ontology refactoring using VDocs refer to [LZ10].

4.6.5 Dependency Graph of Documents

During OMDoc-based collection evolution it is very useful to track dependencies between documents to guarantee the integrity of interlinks. Therefore graphical representation of dependency graphs may be beneficial. For this we have implemented a VDoc that generates such a graph for a given document. As an output the VDoc produces a graph in the DOT language. This is an example of a VDoc application where
4.6 VDoc Use Cases

non-XML content is produced. The content of the VDoc is piped into a TNTBASE plugin (Chapter 6) that produces SVG [FFJ03] from DOT (Figure 4.3).

![Figure 4.3: An Example of a Dependency Graph](image)

### 4.6.6 Bibliography Visualization

As a yet another use case for VDocs that we are dealing with is a KWARC group [Kwa] bibliography visualization. Over the years we have collected a huge number of entries for our own publications and the ones that are cited in our papers. It is rather a common task to get a view on those bib entries depending on the author or a conference. For this the filtering VDoc has been implemented that may take different parameters as criteria for filtering. Later on the content of the VDoc is rendered into HTML using presentational workflows of TNTBASE (cf. Chapter 6). If a mistake is spotted in the HTML presentation, a user may go back to the XML content of the VDoc and fix it on the spot without a necessity to search for a broken entry in the whole bib file. Furthermore, the techniques for generating DOT graphs from a previous use case are used to present a graph of how certain authors, conferences and papers are related (Figure 4.4).
4. SCRIPTING XML DOCUMENTS

Figure 4.4: An Example of Bibliography Visualization

4.7 Conclusion and Further Work

In this chapter we presented the concept of virtual documents and its prototypical but mature and stable realization in the TNTBASE system. VDoc Specs integrate computational facilities into documents like JSP/PHP or \texttt{TeX}/\texttt{LaTeX}, only that VDocs use the versatile and XML-optimized query processing as a computational process instead of relational database lookup (PHP) or general macro expansion in the latter case. Going further with analogues, VDocs as described have very much in common with Java Server Pages (JSP) \[\text{Url}\]. Roughly speaking, JSPs also have “static” parts as HTML elements with embedded “computational” parts in the form of Java code (scriplets). Analogously, in VDocs “static” parts are XML fragments intermixed with “computational” parts in the form of XQuery expressions. In VDocs it is also possible to “import” queries and parametrize them. Similarly, JSPs have a notion of tag libraries: self-contained parametrized units that can be embedded into other JSPs. The goal of both is a clean division of responsibilities.

We view the integration of computation in documents as an enabling technology that explains much of the success and usefulness of the respective approaches, and contend that our VDoc Specs are one way of introducing this to the XML world. We feel that we have just skimmed the practical possibilities induced by VDoc Specs in the use cases discussed in Section 4.6.

For instance, we envision that VDocs can serve as a basis for news generation that are tailored to a particular user and keep track of the news that have been read already. Thus a reader would receive only those topics that are interested for him and has not been explored so far. The targeted VDoc would contain an XQuery that takes parameters specific to a particular user, like interested sections or identifiers of read items. The only part still missing to realize this is a user model for preferences and explored news, but it is a separate problem. The important thing that a single VDoc can satisfy needs of multiple users at the same time.

Note furthermore that the realization of VDocs is not tied to the TNTBASE system – even though version management can profit from VDocs, it is not a prerequisite. In particular, as our implementation is based on XQuery in its core, it should be possible to port it to other XML databases.
4.7 Conclusion and Further Work

In our use cases, the ability to re-use XQuery expressions for different situations and over time has been a crucial ingredient for practical use of VDocs. We therefore anticipate that common XQuery statements will be rolled into extensions for document formats much like macro packages in TeX/LaTeX and thus will create an avenue for user-driven format extensions that may well drive evolution of XML-based formats in the future.

An enabling technology must, of course, also have enabling tools, which is beneficial to have on top of the TNTBase system. One such tool is an editing framework for VDocs. Note that this is non-trivial, since — like their underlying XML formats — VDocs need to be presented to a user in a human-oriented format for reading and editing. Let us consider XHTML as a presentation format. JavaScript frameworks such as JOBAD could be extended in order to inject JavaScript into XHTML that marks up the editable parts of a transformed VDoc with node origins. Special markup will allow that framework to figure out what parts of a presentational document correspond to what parts of the sources from which VDoc content was comprised. The result could be that every “editable” part of rendered XHTML contains a button or a link for editing pressing which results in some popup that shows an editable fragment of an XML document. Pressing the submit button will modify the original content of a VDoc and commit it back to TNTBase. XML sources will be transparently patched that will lead to an updated XHTML version of a considered VDoc. Such an approach will allow a typical user to understand better the meaning of a VDoc (with help of human-oriented presentations) as well as provide interactive means for utilizing the concept of VDoc editing.

---

1 which require specialized expertise and therefore constitute a significant investment

2 Think e.g. of tableofcontents, references, or index elements that abbreviate respective XQueries.
4. SCRIPTING XML DOCUMENTS
5.1 Introduction

XML provides mature facilities for processing and presenting information. The solution developer’s toolkit includes XSLT \[Kay07\] to convert information from one vocabulary to another, XQuery to retrieve persisted information for application consumption, and XHTML \[The02\] and XSL-FO to present information on the screen or in print. These standards have become essential tools for web applications, publishing pipelines, electronic interchange, and many other challenges. Having successfully addressed such fundamental challenges, the XML community can now refine the toolkit. In particular, the XML community can consider whether lessons learned by other communities could have benefits for the XML technology portfolio.

The experience of the Java community with tag libraries offers one such opportunity. Java introduced tag libraries to broaden the appeal of an existing, successful technology, JSP (Java Server Pages) \[Url\]. Widespread adoption exposed a fundamental issue with JSP: mingling presentation and application logic made both kinds of logic more difficult to maintain and prevented specialization. The Java community solved these problems by introducing tag libraries.

In the Java technology stack, tag libraries are function libraries called by embedding a corresponding markup vocabulary (the tags) within documents. The approach is open-ended in that each tag library supplies its own markup vocabulary. The canonical example of a tag library retrieves rows from a database for styling in a document.
5. TAG LIBRARIES FOR XSLT AND XQUERY

In the preceding fragment, the final output does not include the \texttt{taglib:attendee-rows} tags but, instead, \texttt{html:tr} row for each attendee in the dataset processed by the tag library.

The tag library approach introduces a useful division of responsibilities. Tag library developers are responsible for information retrieval and manipulation to produce dynamic documents. Tag document designers are responsible for layout and styling of the dynamic documents. By fostering collaboration between specialists and ensuring, tag libraries improve the overall solution. The tag library approach has provided the basis for comprehensive toolkits such as JSF (Java Server Faces).

The XML technology stack currently lacks a standard mechanism for division of responsibilities. This chapter proposes adapting the Java Tag Library approach by implementing tag libraries in XSLT or XQuery and making use of tag vocabularies in XHTML, XSL-FO, or other XML documents (tag documents). Before introducing our solution, we review some previous initiatives in this area.

5.2 Related Work

Several XML initiatives have addressed integration of information processing and output styling and layout, including:

\textbf{Simplified Stylesheets} XSLT provides Simplified Stylesheets as a method for embedding a stylesheet without templates or functions inside a single result element. Despite freely available implementations, this approach has not seen widespread use, perhaps because information processing is quite limited without templates and functions.

\textbf{Cocoon XSP} XSP \cite{Coc} provides a method for replacing placeholders in XML documents with the results from Java logic using bindings defined with XSLT. Because Cocoon runs in a Java web application environment, the primary focus of XSP is leveraging Java capabilities for producing content within XML documents.

\textbf{XBL} XBL \cite{Xbl} provides flexible aggregation of content fragments and of client resources such as JavaScript methods. While XBL integrates client-side processing, XBL does not directly address styling and layout of content produced by dynamic processing.
5.3 Principal Constructs

RunDMC The general approach of writing stylesheets to bind placeholders within a layout document to dynamically generated content was worked out ten years ago [Xst]. RunDMC [Run] is a recent XSLT-based framework exemplifying the approach. RunDMC matches and replaces placeholders in a layout document with dynamically generated content, potentially parameterizing the dynamic processing with content supplied by the placeholder. While RunDMC shares with tag libraries a focus on the separation of document design from information processing, the burden falls more heavily on the tag library because the binding is maintained in the tag library instead of controlled by the tag document.

The tag library approach differs from these alternatives by supporting a collaboration of equals between the tag document designer and the tag library developer, in particular by supporting callbacks (dynamic content that is constructed at a time chosen and with parameters supplied by the tag library). Callbacks give the document designer full control over styling and layout in ways unforeseen by the tag library developer.

5.3 Principal Constructs

In the tag library approach, the tag document contains the following constructs:

Document vocabulary A tag document makes use of a vocabulary unrelated to tags, typically a presentation vocabulary such as XHTML or XSL-FO. The other constructs of the tag document mixed with the document vocabulary to produce the dynamic content for the document. Because the tag constructs are not part of the document vocabulary and because the dynamic content is not present in the source, tag documents are typically well formed rather than validated prior to processing.

Expression language Java Tag Library technology supports a simple expression language for producing attribute values or element content. These expressions closely resemble XQuery enclosed expressions and XSLT attribute value templates. XPath provides a natural expression language for addressing and manipulating XML content and, thus, is a reasonable choice to fill this role. Some scenarios for use of expressions include

- Evaluating variables
- Manipulating string, numeric, and date values
- Extracting values from XML documents
- Copying portions of XML documents

Basic Statements Java Tag Library technology supports basic statements for capturing a value or the result of an expression in a variable, for a conditional part of the tag document, for a repeated part of a tag document, and so on. These
5. TAG LIBRARIES FOR XSLT AND XQUERY

Statements have a close equivalent in the XSLT statements supported by Simplified Stylesheets. Examples include xsl:copy-of, xsl:choose, xsl:for-each, xsl:if, and xsl:variable. (The xsl:copy-of statement has a more intuitive result than the xsl:value statement, which can be achieved through use of the XPath string() or data() functions.) Instead of introducing an entire new vocabulary, a reasonable approach is to follow the lead of Simplified Stylesheets and support the subset of XSLT statements that can be embedded in result content.

The actual need for conditional and looping statements in the tag document can be eliminated through tags that assess conditions and repeat on behalf of the tag document.

Tags elements A tag element calls a function in the tag library with parameters supplied by the tag document and insert results as dynamic content within the tag document. The tag element is bound to a function by having the same QName. In the XML environment, parameters can include both atomic values and XML elements or other nodes.

Handlers Java Tag Library technology has a key feature (known in Java as the tag body), which is a content fragment supplied by the tag document but parameterized by the tag library. In effect, the tag body is a callback from the tag library function. The tag body is also a closure with respect to the tag document in the sense that variables in scope within the tag document can be evaluated in the tag body.

XML poses some challenges that capabilities not offered by the Java tag body. In particular, a single callback is not enough to process an XML tree structure. XSLT has template match rules and XQuery has typeswitch expressions for this precise reason. To process XML content with a tag, different dynamic content must be produced for different elements. To highlight this distinction, we have chosen the term handler.

5.4 Vocabulary Syntax

As was mentioned above, one of the prime goals of tag libraries is a division of responsibilities. Therefore, we will structure this section into two subsections, each of which describes the syntax needed for a different audience: tag library developers and tag document designers. The latter audience has the processing capability for complete control over style and layout. Thus, we are seeing the true separation of concerns because the tag library developer understands information processing but need not embed any style or layout. Tag libraries may be implemented in XQuery or XSLT whereas tag documents always have the same syntax, adopting statements from XSLT Simplified Stylesheets.
5.4 Vocabulary Syntax

5.4.1 Tag Libraries

First of all, the main namespace for tag libraries is http://code.google.com/p/xst-tag/ that we will refer by a prefix xst throughout the chapter. Tag libraries provide a number of tag definitions that can be used by tag documents. Often, tag libraries may want to refer to other XSLT stylesheets or XQuery modules in order to utilize existing templates or functions. Thus the import statement must be supported. Essentially, a tag definition consists of two major parts: a list of accepted parameters, which are supplied by a tag call in a tag document, and a body that may contain XML nodes, XSLT or XQuery expressions, and calls to handlers (which are special type of tag definition parameters). The tag definition executes an action on behalf of the tag document, typically generating content for the tag document to lay out or style.

XSLT and XQuery implementations of tag libraries are conceptually similar but differ, however, in syntax. Let us consider the peculiarities of both languages for implementing tag libraries.

5.4.1.1 XSLT Tag Libraries

XSLT Tag libraries have a root element xst:stylesheet. Its children are XSLT import statements, a list of tag definitions, and XSLT templates or functions applied or called during processing of a tag definition. Every tag definition:

- Is declared with an xst:tag element having a name attribute. The tag name must be a QName, e.g. 
  \[<xst:tag name="pref:myTag">...<xst:tag>\], where
  prefix pref is bound to some namespace, and myTag is the name of the tag. Each
tag library typically has a single namespace for all tags provided by the library.

- can have parameters with the usual atomic value or node types but also with
  the handler type. A handler type resembles an XPath 2.1 function signature but
  requires a name for each parameter of the handler. The handler is required or
  optional on the tag call based on whether its signature has a required or optional
  return value. Similar to other XSLT parameters, a tag’s handler parameter may
  contain a default handler definition.

- has a tag definition body after all of the tag parameters. Within the tag definition
  body, the handler definition passed on the tag call can, in turn, be called with
  an xst:call handler="$handler-name" statement or assigned, evaluated, or
  passed within the tag library similar to an XPath 2.1 function object.

A tag library can define a handler only in the default implementation context (and
not, for instance, through assignment to a variable).

Further in this chapter we will use a book store example that seems to crop up
quite frequently in XML technologies. For simplicity we consider a flat book structure
in XML documents. Assume that we have multiple XML documents that contain a
list of books where each of those may contain the following information: a title, genre,
multiple authors, a description, a publish date and price. For example:
In regard to a book store use case, an example of an XSLT tag library with a single tag definition may look like this:

```
<xst:stylesheet version="0.2"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
  xmlns:fn = "http://www.w3.org/2005/xpath−functions"
  xmlns:xst = "http://code.google.com/p/xstag/"
  xmlns:tag = "http://code.google.com/p/xstag/books/tags">
  <xst:tag name="tag:books" as="element()∗">
    <!-- tag definition parameters -->
    <xst:param name="years" as="xs:integer∗" />
    <xst:param name="onBook" as="handler($title as xs:string, $authors as xs:string+)" as element()?" >
      <!-- default handler implementation -->
      <tr>
        <td><xsl:copy−of select="$title" /></td>
        <td><xsl:copy−of select="fn:string−join($authors, '; ')" /></td>
      </tr>
    </xst:param>
  </xst:tag>
</xst:stylesheet>
```
The above tag library contains only one tag definition that is bound to namespace http://code.google.com/p/xstag/test/tag via the prefix tag. The tag books takes two parameters: a list of integers that denote years of publications, and a handler parameter that, in turn, takes a title and a list of authors. If a tag caller do not provide his own handler definition then the default implementation from the tag definition will be used: in our case the default implementation returns an HTML table row that contains two columns: a book title and concatenated list of authors.

The handler definition body uses XSLT to retrieve books based on supplied years (this part is omitted for brevity) and then, for each retrieved book element, the tag definition body calls a handler definition with the parameters of the handler.

5.4.1.2 XQuery Tag Libraries

For XQuery tag libraries we propose to extend syntax of XQuery so it looks native to tag library developers. An XQuery tag library module has tag definitions, which are:

- declared with the tag keyword instead of the function keyword.
- can have parameters with the usual atomic value or node types but also with the handler type. A handler type resembles an XPath 2.1 function signature but requires a name for each parameter of the handler. The handler is required or optional on the tag call based on whether its signature has a required or optional return value.
- has a tag definition body analogous to the function body. Within the tag definition body, a handler can be called with handle statement or assigned, evaluated, or passed within the tag library.

An example from Section 5.4.1.1 can be rewritten to:

```xml
<xs:element name="books">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="book" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="handler-type" type="xs:string"/>
    <xs:attribute name="xs:string"/>
  </xs:complexType>
</xs:element>
```

This syntax is conceptual. For a practical proof of concept, it may be desirable to use an XML syntax for the tag signature and handler call to simplify implementations of tag processors.
5. TAG LIBRARIES FOR XSLT AND XQUERY


In the example above we declare the tag library with the taglib keyword analogously to a library module declaration. Every tag in the tag library should be in the tag library namespace. We also defining the xst namespace as well as we import other XQuery library modules.

Unlike XSLT tag libraries, handler parameters of tag definitions in XQuery tag libraries may not have the default implementation. This decision is consistent with the XQuery environment in which parameters cannot have default values.

The tag definition body contains calls to the function from the imported module, and also calls a handler similarily to inline function calls in XQuery 3.0 [Rob+10]. XPath expressions are used to pass the right parameters to the handler.

5.4.2 Tag Documents

A tag document is an XML document that contains calls to tags defined in one or more tag libraries. A tag document can call a tag provided by a tag library with an element that has the same QName as the tag. The element making the tag call can have:

- Subelements or attributes that supply parameters of the tag.
- Subelements that supply handler definitions for handler parameters of the tag. When processing a handler, the tag call can pass parameters to the handler. The handler definition can evaluate the parameters passed by the tag as part of its content.

A tag document can only define handlers as part of a tag call. A tag call can be nested within a handler.

If a tag has only one handler or only one parameter with a node type (but not both), the tag element can omit the subelement container and contain the node or handler content directly.

Tag documents can use the following elements that are a superset of XSLT Simplified Stylesheets:

- a tag library reference xst:import href="..." for importing a tag library. Both XSLT and XQuery tag libraries can be imported without making a difference for a tag document designer.
- branching and looping: xst:choose/xst:when/xst:otherwise, xst:for-each, xst:if
- variables (constants) and evaluation: xst:copy-of, xst:param (globals only), xst:variable
5.4 Vocabulary Syntax

- explicit invocation of tags using `<xst:call tag="tag:foo"/>` and supplying parameters and handlers with the `xst:with-param/xst:with-handler` elements

- `xst:include` another XML Style Tag module at the position

Even though many constructs above have the same meaning and local name as constructs from XSLT Simplified Stylesheets, tag documents are not supposed to lean towards XSLT. It just happened to be a convenient way to express most used constructs using very well adopted XSLT syntax but in different namespace. Using only the `xst` namespace allows tag document designers not be forced to remember which namespace to use for what constructs.

It is also worth mentioning that variable scope is limited to following siblings and their descendants. Handlers can refer to variables in scope. An example of a tag document:

```xml
<html xst:version="0.2"
xmlns:xst = "http://code.google.com/p/xstag/"
xmlns:fn = "http://www.w3.org/2005/xpath−functions"
xmllns:tag = "http://code.google.com/p/xstag/books/tags"/>
<!−− Importing a books tag library −−>
<xst:import href="BooksTaglib.xstag"/>
<xst:variable name="years" select="(2010, 2009, 2008)"/>
<head>
<title>Books for <xst:copy−of match="$years"/> years</title>
</head>
<body>
<table>
<xst:copy−of select="$years"/>
<tag:books>
<tag:years>
<xst:copy−of select="$years"/>
</tag:years>
<tag:onBook xst:handles="$title, $authors">
<tr>
<td>{$title}</td>
<td>{fn:string−join($authors, '; ')}</td>
<td>{fn:count($authors)}</td>
</tr>
</tag:onBook>
</tag:books>
</table>
</body>
</html>
```

The example above creates an HTML document with a table of books for particular years. This tag document makes use of a tag library presented in Section 5.4.1.1. First of all we import the tag library using the import statement: `<xst:import href="BooksTaglib.xstag"/>`. In order to use tags from that library, the namespace for the tag
call element (tag:books) has to match the namespace for the tag definition element but does not have to have the same prefix.

Our tag has two parameters (refer to Section [5.4.1.1] for more details): a sequence of publishing years and a handler parameter that takes a string parameter for a title and a sequence of strings for the list of authors. Every non-atomic parameter passed to a tag definition should be a child of a tag call with the same tag namespace and the name of a corresponding parameter. In our case the tag:books tag definition takes parameters with the names years and onBook (a handler definition). Therefore in the tag call we have the tag:years and tag:onBook elements. Note that we could avoid providing a handler definition since a tag definition in the tag library already has a default implementation. However, in this case we decided to "override" the default handler definition and supply a third column in a row that denotes the number of authors of any particular book (in addition to the title and concatenated list of authors of the book).

It is worth noting that a tag document also supports attribute value templates and element content templates for XPath expressions. We can use curly brackets for embedded expressions as in XQuery (e.g. <td>{$title}</td>).

Note the xst:handles attribute on the handler parameter element. The attribute identifies the set of parameters that the handler expects the tag definition to pass. If a tag definition takes only the handler parameter and the tag call uses the shortcut to dispense with the parameter container element then the xst:handles attribute should be provided on the tag call element itself (i.e. tag:books in our example).

Whereas a tag processor can (from looking at both a tag library and a tag document) define what parameters that are passed are XML element parameters or handler definitions, there are some advantages to having the xst:handles attribute:

- By looking at the tag document, anyone can distinguish handlers from node parameters.
- A tag processor can compare the parameters expected by a handler definition with the parameters declared formally for the handler by the tag definition and detect errors if, for instance, a change in the tag library invalidates the tag document.
- A tag processor can also check to confirm that every variable reference in the tag document is in scope either within the tag document or as a parameter of the handler definition.

Theoretically it is not necessary to have the xst:handles attribute; however, we encourage it for the benefits pointed out above. To minimize the formality for the tag document designer and maintenance generally, the type information for each parameter would be declared only in the tag library. If the tag definition passes a parameter to the handler definition that the handler definition does not need, the handler definition can simply ignore the parameter. In particular, the xst:handles attribute does not have to list unused parameters passed by the tag definition. Moreover, order of parameters in the xst:handles attribute does not matter: XSLT does not have a concept of
parameter order, and even in case of XQuery, a handler definition only cares about which parameters are in scope.

The `xst:handles` attribute does not control the handler call from the tag definition to the handler definition. Instead, the handles attribute is similar to a C external function signature for a function defined in a library.

Additionally, with regard to tag documents in general, it is worth mentioning several things:

- if we had to pass an atomic parameter such as a name of genre in the tag call, then we can utilize attributes of the tag call. For example, if the `tag:books` accepted a parameter `genre` as a string value, then we would write `<tag:books genre="Fantasy"> ... </tag:books>`. However, it is not necessary: an atomic value could be passed by a subelement, given that the parameter in the tag definition identifies the data type. On the other hand, a node could be passed by an attribute if it is supplied by an expression that evaluates to the node. However, handlers cannot be passed as attributes since we need to process them in context.

- If a tag takes only one handler parameter or one node sequence parameter (but not both), we could omit a parameter container (see Glossary 5.8). For example, if a `tag:books` needed only one handler, then we could call the tag in the following manner:

```
<tag:books xst:handles="$title, $authors">
  <tr>
    <td>{$title}</td>
    <td>{fn:string-join($authors, '; ')}</td>
    <td>{fn:count($authors)}</td>
  </tr>
</tag:books>
```

Note that in this scenario the `xst:handles` attribute belongs to the tag call element itself since there is no parameter container for the handler in this case.

However, in case when a result of a certain tag should be passed as the only parameter to another tag with the same namespace, then the usage of a parameter container is mandatory. Theoretically, it should not be necessary in all cases, but, generally, such a constraint reduces possible ambiguity.

- Results of tag calls can be bound to some variables and those variables can be reused in other tag calls. See Section 5.5 for the more elaborated books example.

### 5.5 Complete Book Example

Here we will provide a complete books example that utilizes different techniques available in the style tags proposal.
5. TAG LIBRARIES FOR XSLT AND XQUERY

A tag library with multiple tags:

```xml
<xst:stylesheet version="0.2"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fn="http://www.w3.org/2005/xpath−functions"
xmlns:xst="http://code.google.com/p/xstag/"
xmлистag="http://code.google.com/p/xstag/books/tags">
  <xst:tag name="tag:booksByYears" as="element() ∗">
    <xst:param name="years" as="xs:integer ∗"/>
    <xsl:copy-of select="doc(...)//book[year=$years]"/>
  </xst:tag>

  <xst:tag name="tag:htmlBooksByGenre" as="element() ∗">
    <xst:param name="genres" as="xs:string ∗"/>
    <xst:param name="books" as="element() ∗"/>
    <xst:param name="onTitle" as="handler($title as xs:string) as node()"/>
    <xst:param name="onBook" as="handler($title as xs:string, $authors as xs:string+, $genre as xs:string) as element()?"/>
    <tr>
      <td><xsl:copy-of select="$title" /></td>
      <td><xsl:copy-of select="fn:string−join($authors, '; ')" /></td>
      <td><xsl:copy-of select="$genre" /></td>
    </tr>
  </xst:tag>

  <xst:for-each select="$books[fn:string(genre) = $genres]">
    <xst:call handler="$onBook" name="title">
      <xst:with−param name="genre" select="fn:string(title)"/>
    </xst:call>
    <xst:with−param name="authors" select="author/fn:string(.)"/>
    <xst:with−param name="genre" select="fn:string(genre)"/>
  </xst:for−each>
</xst:tag>

<xst:tag name="tag:dateTag">
  <xst:param name="date−handler"/>
</xst:tag>
```

118
5.5 Complete Book Example

as="handler($current-date as xs:date) as element()"/>

<xst:call handler="$date-handler">
  <xst:with-param select="fn:current-date()"/>
</xst:call>
</xst:tag>

</xst:stylesheet>

A tag document that uses all of the declared tags above might look like:

```xml
<html xst:version="0.2"
  xmlns:xst = "http://code.google.com/p/xstag/"
  xmlns:fn = "http://www.w3.org/2005/xpath-functions"
  xmlns:tag = "http://code.google.com/p/xstag/books/tags">
  <xst:import href="BooksTaglib.xstag"/>
  <xst:variable name="years" select="(2010, 2009, 2008)"/>
  <xst:variable name="genres" select="('Fantasy', 'Cooking', 'Kids')"/>
  <xst:variable name="booksByYear">
    {"years"}
  </xst:variable>
  <head>
    <title>Books for <xst:copy-of match="$years"/> years</title>
  </head>
  <body>
    <xst:tag:dateTag xst:handles="$current-date">
      <div class="tableClass">
        <p>Generated on <xst:copy-of select="$current-date"/></p>
        <table>
          <thead>
            <tr>
              <th><xst:htmlBooksByGenre><xst:genres>{$genres}</xst:genres></xst:htmlBooksByGenre></th>
              <th><xst:books>{$booksByYear}</xst:books></th>
            </tr>
          </thead>
          <tbody>
            <tr>
              <td><xst:onTitle xst:handles="$title"></td>
              <td>
                <xst:onBook xst:handles="$title, $authors">
                  <td>{$title}</td>
                  <td>{fn:string-join($authors, '; ')}<td>
                  <td>{fn:count($authors)}</td>
                </xst:onBook>
              </td>
            </tr>
          </tbody>
        </table>
      </div>
    </xst:tag:dateTag>
  </body>
</html>
```
5. TAG LIBRARIES FOR XSLT AND XQUERY

The example above demonstrates the flexibility of the tag approach and emphasize the following features: constant variables, assigning a result of a tag call to a variable, passing atomic, sequence, element and handler parameters, default handler definitions, multiple handler parameters, calling one handler from another in the tag definition, parameters with and without parameter containers, using one tag call within another tag call, using XSLT-like calls (i.e. \texttt{xst:copy-of} in these examples) outside the tag call.

5.6 Tag Processors and Implementation

The implementation of the tag library approach is responsible for producing output document based on the tag document and imported tag libraries. We call the implementation a \textit{tag processor} and an output document a \textit{result document}.

Tag processors for XSLT and XQuery are conceptually similar (see Figure 5.1). Let us emphasize the most important aspects of how a tag processor produces a result document:

- The tag processor descends the XML tree of a tag document, defining variables in scope within a branch. Those variable are outside any tag call and, essentially, are constants.

![Figure 5.1: Tag Processing Workflow](image)

120
5.6 Tag Processors and Implementation

- When the tag processor reaches a tag call, it evaluates any expressions for atomic value and node parameters including expressions embedded within literal elements and replaces each expression with its result before calling the tag. We call this phase the time of the call. Conceptually, any variables defined outside of the tag call and referenced within a handler are also replaced by their values at the time of the call. That is, a handler is a closure with respect to the tag document outside the tag call. The xst:handles attribute allows the tag processor to easily distinguish handler parameters from variables which should be substituted with their values.

- During the time when the tag processor starts executing the tag, when the processor reaches the handler call within the tag definition, the processor passes the parameters of the handler call and evaluates the expressions in the handler, replacing each expression with its result. This phase is called the time of handler processing.

In the XSLT implementation, a tag call is preprocessed to a call passing a temporary XML document that has a subelement for each handler and a subelement capturing the state of each variable defined outside the call but referenced in a handler (excluding parameters, which by definition are global).

A handler is preprocessed to a match template for the handler subelement, assigning local variables from the state subelements for each out of scope variable that is referenced in the handler.

A tag template is preprocessed to a named template with a single parameter for the temporary XML document and local variables for the handler subelements that (when not provided by the temporary document) default to the content of the handler parameter.

A handler call is preprocessed to apply on the handler subelement of the call document.

If the XQuery processor supports closures, a preprocessor can convert a handler into a closure that is passed as part of the tag call. The preprocessor can treat the handlers in a tag library as a function item parameter.

If the XQuery processor supports function items, a preprocessor can convert a handler into function and pass the function to the tag. The preprocessor can also capture the state at the point of call and restore the state within the constructed function by passing a temporary XML document (as in the XSLT implementation).

Otherwise, a preprocessor can generate a copy of the tag library for each tag document, generating a function for each handler from the tag document and passing and restoring the state as a temporary XML document.

Rather than introducing a set of statement elements in a different namespace that mean the same thing, it makes more sense to adopt the subset of XSLT statements that provide an equivalent to the statements in a Java Tag Library. That way, it is:

- Easy for XSLT developers to support tag document designers since the syntax should be familiar.
5. TAG LIBRARIES FOR XSLT AND XQUERY

- Easier for tag document designers to develop tag libraries if needed.

To summarize, the reason for adopting a subset of XSLT is not because of an intent to favor XSLT over XQuery but applying the lessons learned from Java Tag Library.

5.7 Conclusion and Future Work

Java Tag Library technology has seen wide adoption as a templating strategy because of its separation of concerns between presentation and data manipulation. In particular, because a tag document establishes bindings on components, a tag library removes the need to write binding logic to add or change the libraries used in the document.

This chapter demonstrates the feasibility of adapting the Java Tag Library approach for use with XML technologies. Tag libraries can support a set of data retrieval and manipulation functions as well as UI components. Tag documents can support these functions by embedding tags within a presentation vocabulary such as XHTML or XSL-FO. In particular, tag documents can pass handlers to tag libraries for parameterization of document content by the tag library. Tag library processors can be written in either XSLT or XQuery.

Tag library developers need to provide guidance to tag document designers on the use of the library. To make that easier and more consistent, a mechanism similar to JavaDoc is important for maintaining documentation source within tag libraries and generating viewable documentation. Existing projects such as XSLTdoc [Xsl] or xqDoc [Xqd] may be adaptable for this purpose. The problem and possible solutions require more investigation.

In AJAX scenarios, documents can present changing data on the browser client without reloading the document from the server. While servers typically have several options for XSLT and XQuery processors, many browser clients have limited XSLT and XQuery processor options. The tag library approach suggests a potential solution for this challenge. A basic tag processor could be written in JavaScript, executing some of the tag calls as AJAX requests that send parameters and handlers to an XSLT or XQuery tag processor on the server and returning the result for insertion into the client document. In addition, JavaScript tag libraries could allow tag documents to perform appropriate data manipulation on the client. Such environment might benefit from a simpler expression language that could be implemented in either XPath or JavaScript. The specifics for processing some tag calls on document request, some tag calls entirely on the client, and some tags calls through AJAX requests requires more investigation.

Tag libraries have some implications for the XQuery and XSLT standards. Where the processor supports function objects with closure over variables in scope for the function definition, handlers become trivial to implement. The current XQuery draft envisions such function objects. The tag library approach suggests the value of such function objects for XSLT as well. On a separate issue, Simplified Stylesheets have never seen widespread utilization, perhaps because the standard does not provide the document with sufficient processing capability. Tag documents share many features
with Simplified Stylesheets but provide much more capability. The tag library approach could be standardized as a revision of Simplified Stylesheets shared by both XQuery and XSLT.

While future investigation may improve the tag library approach, it already has some clear benefits. Tag libraries can be implemented in either XSLT or XQuery, enlarging the environments in which tag libraries can be deployed as well as the pool of potential tag library developers. A tag document can be processed in both environments, allowing tag designers to provide designs for either kind of environment without having to learn a new syntax. Through the contract of the tag signature – the parameters and handlers and return type passed to the tag – contributors can apply their distinct expertise to collaborate for information processing and presentation.

5.8 Glossary

tag document
A document that mixes a static XML vocabulary (such as XHTML or XSL-FO) with XPath expressions, XML Simple Tag statements, and tags from one or more tag libraries to produce dynamic documents.

tag library
A function library (typically implemented in XSLT or XQuery) that produces XML content for a tag document.

simple tag statement

tag call
An element in a tag document processed as a function call to a tag definition with the same QName, supplying parameters with its attributes and content and replaced by the return value from the tag definition.

tag definition
A function provided by a tag library that processes parameters to produce XML content.

tag parameter
An atomic value, XML node (typically one or more elements), or handler definition passed by a tag call to a tag definition to control production of the XML content or to supply content for insertion into the XML content.

tag definition body
Content of a tag definition followed after declaration of all tag parameters. A tag definition body defines the logic how a tag call will be executed.
5. TAG LIBRARIES FOR XSLT AND XQUERY

parameter container
A subelement of the tag call supplying a tag parameter with its content; the subelement has a namespace prefix that is the same as the prefix of the tag call and a local name that’s the same as the name of the tag parameter.

handler definition
An XML fragment that is passed as a tag parameter and that is processed with values generated during execution of the tag definition; a callback.

handler definition call
Processing of a handler definition with values during execution of a tag definition.

handler definition parameter
A value passed by the tag definition to a handler definition.

tag processor
An implementation of the tag library and tag document functionality that takes them as an input and produces a result document.

result document
A document produced by a tag processor based on tag libraries and a single tag document that import those libraries.

5.9 Comparison to Virtual Documents

The VDocs and tag libraries concepts share a lot in common. We compare these two technologies based on the following criteria:

C1 Expressiveness in defining VDocs and tag libraries provide different means to define the result documents. The VDocs concept provide a small set of instructions for XQuery inclusion, but at the same time a VDoc Spec can contain additional parameters and XQuery queries, either explicitly or linked to another resource via a URI. In turn, the tag libraries approach provides a rich set of instructions that is very similar to the XSLT syntax, and is more powerful. Analogues of parameters in VDoc Specs are \texttt{xst:variable} statements which also can be external, that is not defined in a tag document or library but are provided externally. Unlike VDocs it is impossible to import queries from external resources in the current version of the tag libraries approach.

Additionally, tag libraries also support the XSLT language. Nothing prevents from using XSLT in the VDocs concept, but this question needs more investigation since VDocs have been developed in the scope of XML-native databases which typically do not support XSLT.
C2 **Pre-processing and execution** The results documents are obtained differently. A VDP operates like an interpreter having a VDoc Spec as an input and producing a result document. On the contrary, a tag processor is a compiler that first compiles a tag document to a set of main/library XQuery modules (or XSLT stylesheets) and then an XQuery/XSLT processor executes a compiled result. On the other hand, a VDP can be implemented as a compiler as well. The advantages of the compiler approach is that a compiled version of a tag document can be cached until the corresponding tag documents and imported tag libraries have been changed and executed more efficiently by skipping the compilation phase. Moreover, an interpreter approach requires a processor-specific XQuery evaluation functions or an implementation of external functions, which makes an interpreter code dependent on the XQuery processor.

C3 **Division of responsibilities in document creation** In VDocs division of responsibilities is done in a simpler manner since the expressiveness as discussed in item C1 is lower. It is done by means of inheriting VDoc Specs and overriding parameters and queries. Also an XQuery query can be just imported into a VDoc Spec from another resource. The tag libraries approach offers a cleaner separation of concerns by allowing people with different expertise either program tag libraries or reuse tags in tag documents.

C4 **Integration into a file system and materialization** This topic has not been discussed for tag libraries but the same approach for VDocs (see Sections 4.2.4 and 4.3.3) can be adopted.

C5 **Querying** Querying for VDocs can be done via an XQuery external function that interprets a VDoc Spec and produces a result document. It also can be done on the materialized snapshots of VDocs. In case of tag document either an external XQuery function should be provided that generates the result document on the fly, or querying can also be performed on the materialized counterparts of tag documents. In the former case we became dependent on the XQuery processor.

C6 **Editing facilities** An ability to propagate changes back to original sources has not been discussed for tag libraries. However, the author envisions that the same approach with origin markers (cf. Section 4.2.6) can be adopted.

C7 **Applicability to versioning** This topic has not been considered for tag libraries since it was not integrated into TNTBase (or some other VXD) yet. However, if an XQuery model is extended to address a temporal aspect it is straightforward to utilize extended XQuery language capabilities in tag libraries.

The tag library technology overall is more advanced than VDocs. Integration of the VDoc concept into a VXD has been considered much more in detail. A strong advantage of VDocs is the support for editing which is enabled by the underlying VDP.
5. TAG LIBRARIES FOR XSLT AND XQUERY
Part IV

Evaluation
So far we considered theoretical aspects and engineering techniques of how we can build a versioned XML database that we discussed in Part II. The TNTBase system has been implemented to show the feasibility and applicability of the discussed approaches. VDocs and tag libraries approaches and their realization in TNTBase further demonstrated how XML-based workflows can be enhanced and we showed this in Part III. Theoretical concepts gain more value when supplemented with use cases that demonstrate the practicality of the proposed solution. The topics discussed so far will be evaluated from three perspectives: (i) adaptation and extending to a certain XML language domain (Chapter 6) (ii) simplicity of usage and powerfulness of features (Chapter 7) (iii) efficiency and reliability (Chapter 8). Over past 3 years the TNTBase system has matured to reliably store gigabytes of data – indeed we never experienced loss or data corruption. We used TNTBase for different applications that required utilization of mixed content from more than 10 repositories.

To show that the practicability and extensibility of the concepts and the implementation, we will start with a success story of extending TNTBase to the MKM field by adapting the system to the OMDoc domain since it was the XML format that we mostly experimented with.
Chapter 6

Adaptation to the Specific Domain

6.1 Introduction

In Chapter 3 we discussed the approach of how to create a general-purpose versioned XML database leveraging the XeR concept described in Chapter 2. The VXD concept is intended as a generic storage solution for applications that makes extensive use of XML as the underlying storage model. However, most such applications are tailored to specific features of their respective document formats. For instance, DocBook [WM08] manuals rely on format-specific stylesheets for web presentation and printing, which only work if all documents are valid instances of the DocBook format. This is also the case for other applications; some even combine multiple document formats and thus need to support multiple validation and presentation pipelines. Furthermore, even homogeneous collections may contain documents in various versions of the underlying formats: indeed the versioning capabilities of the VXD approach should enable the management of long tails of legacy materials.

In our experiments with the TNTBase system, we have found that many of the format-specific functionalities of web applications layers can be performed by the TNT-Base system if we add format-specific interfaces for validation, aggregation, and presentation. This suggested that those interfaces can be generalized with respect to a particular VXD.

We will consider those interfaces as a separate layer in a VXD and will call a VXD augmented with it a VXD(F), since the services are parametric in the document format F. This extension leads to a refined information architecture of VXD-supported web applications as can be seen on Figure 6.1 where the generic VXD(F) layer takes
over format-specific functionalities and thus decreases the effort for developing web applications.

In this chapter we discuss how a VXD implementation can be extended with validation, (human-oriented) presentation and XML differencing by drawing a successful example for our group’s OMDoc format \[\text{Koh06b}\] where \(\mathcal{F}\) = OMDoc. However, we will be slightly more abstract so as to arrive more intuitively to how any VXD can be adapted to any other XML document format. The implementation of TNTBase(OMDoc) is based on the extensive experience of managing a collection of more than 4000 OMDoc documents, ranging from formal representations of logics to course materials in a first-year computer science course.

6.1.1 Motivation

Validation and Interface Extraction  One of the salient features of XML as a representation format is its well-established methods for document validation: most document management workflows take advantage of this to check grammatical constraints on documents by schema validation which simplifies further document processing. In some practical case schema validity is too strong a restriction (e.g., during document authoring or format migration), so the level of validity should be configurable. Moreover, many document format constraints – e.g., inter-document link consistency or the absence of link cycles – cannot be checked even by modern schema languages. Therefore it is necessary to provide an API for custom validation modules that can verify that high-level document collection integrity constraints are met, if later processing steps require them.

Both high-level validation and further document processing often rely on specific information about other parts of the document collection. To support these processes incrementally, VXD(\(\mathcal{F}\)) should support the extraction and indexing of such information both on requests and commits. This approach is analogous to the “separate compilation” paradigm in programming, where declared methods and their types are extracted at compile-time into “signature” files that are used when compiling other files. A prominent example in the XML arena is the extraction of RDF \[\text{Bec04}\] triples for the use in query engines from RDFa-annotated documents. As the specific information is format and application specific, we speak of interface extraction for the purposes of this chapter. VXD(\(\mathcal{F}\)) should enable a user to configure interface extraction at commit time; in our experience, this is a crucial ability to make high-level validation tractable.
**Compilation, Browsing and Cross-Referencing**

Eventually, the purpose of most document collection applications is to enable humans and machines access to processed forms of the document collection. Technically, this usually involves the transformation into specialized presentation-oriented formats for machine (e.g., programming languages) or human (e.g., PDF generation) consumption. Depending on transformation properties and the rate of document collection changes, it can be more efficient to execute the transformations when a document is committed to the repository. Then we can employ a process analogous to interface extraction. It can also happen when a document is requested or its presentation is lazily generated at the background when system load is low. All workflows need to be supported in VXD(\(\mathcal{F}\)), and in these cases the documents are automatically served in addition to their sources. VXD(\(\mathcal{F}\)) should provide an enhanced browsing interface for cases where document post-processing defined so as to facilitate human data consumption. A VXD(\(\mathcal{F}\)) architecture will make it possible by providing higher-level presentational services such as navigation bar, cross-referencing, and in-place expansion of links, if it is defined which language features constitute document fragments and links to documents or document fragments.

### 6.2 Towards TNTBase(OMDoc)

In our experiments we have distinguished two major types of XML document processing:

- **Validation** A binary function that determines whether a document is correct according to some criteria. Often validation frameworks provide some form of validation results after success. One of the prominent examples is a result of XML Schema validation – Post Schema Validation Infoset (PSVI) defined in the XML Schema W3C Recommendation [GSMT09].

- **Presentation** Aggregation/extraction of auxiliary presentational information from a document or a set of documents [ZKR10].

We call a process of validation or presentation generation an \(\mathcal{F}\)-process or a process for simplicity. We will see that both validation and presentation processes generate a result, and a validation process additionally produces a boolean value. We call it an \(\mathcal{F}\)-result. We can treat both types of processes likewise given that a presentation process generates a positive boolean value. A validation result may also be empty. For instance, DTD validation just reports whether an XML document is valid or not.

In our experiments we concluded the following:

- Validation of OMDoc documents based on some schema validation language is needed. For old OMDoc language versions we have a DTD, whereas for newer versions we have a RELAX NG schema while for other documents formats we may want to use XML Schema, Schematron [Sch], or the like. Although it is common for XML databases to support DTD and XML Schema, other XML validation languages must be supported outside an XML database, if required.
6. ADAPTATION TO THE SPECIFIC DOMAIN

- Results of F-processes are typically XML documents themselves. Having them persistent in TNTBase gives us necessary data locality, history preservation and querying possibilities that had to be exposed by performing some amount of engineering work. Versioning of F-results has not been crucial for us.

- We want to have control over invalid XML documents. In our use cases it may be acceptable to have invalid (but well-formed) XML documents in TNTBase. In other cases invalid documents should never be persistent in TNTBase. We can think of development versus release workflows.

- Both manual and automated ways of performing validation/presentation are required. E.g. we validate OMDoc documents upon commit, and receive XHTML representation upon requests.

- Chaining of F-processes is required. E.g. validating against a RELAX NG schema, and in case of success extracting RDF-triples and persisting them in TNTBase for later processing and retrieval. The XProc [WMT10] language served us a model in this respect since it was intentionally designed for supporting XML pipelines. Data interchange between processes in pipelines is done by persisting F-results in TNTBase.

- We must distinguish between pre-commit and post-commit semantics. Failures occurred in the pre-commit phase must result in the failed commit whereas problems encountered in the post-commit phase must only generate warnings for a user. E.g. basic validation in many cases must be “strict” and disallow commits of invalid XML documents, whereas more sophisticated validation may tolerate certain problems in the documents.

Although we experimented mostly with OMDoc we consider the observations above quite general and applicable to many document formats. Therefore we envision that the work performed for adapting to the OMDoc domain can be beneficial for other domains as well.

We experimented with F-results that are XDM instances. In practice, this is not a significant restriction since many non-XML formats can be presented as strings which are also a part of an XDM. For example, a JSON string can be mapped to the type xs:string. Also many XQuery processors provide utility functions to map JSON or HTML to XML. For example, XQilla has the \texttt{xqilla:parse–json($xml as xs:string?) as element()}? function for that purpose.

Let us give a formal definition of an F-process in the fixed VXD \texttt{XML}.

\textbf{Definition 6.1 (An F-process)} An F-process is a function $\mathcal{P} : N \times \text{xs:string}* \times \text{item()}* \rightarrow \text{xs:boolean} \times \text{item()}*$. For the XDM sequence of parameter names $p$ and the XDM sequence of parameters $s$ an F-process with a process name $n$ produces a pair of a boolean value and a sequence of XDM instances: $\mathcal{P}(n, p, s) = (b, r)$. The numbers of parameter names and parameters must match. $\mathcal{R} = \text{xs:boolean} \times \text{item()}*$
6.2 Towards TNTBase(OMDoc)

is the set of $\mathcal{F}$-results. We call $b$ an $\mathcal{F}$-result boolean value, and $r$ an $\mathcal{F}$-result sequence.

For every VXD we are free to choose the names of $\mathcal{F}$-processes, and $\mathcal{F}$-processes must freely define parameter names and parameters it “accepts”. We will need parameter names when we want to fix some of the $\mathcal{F}$-process parameters, e.g. a RELAX NG schema location within a repository.

Example 6.1 (RELAX NG Validation) Assume that we have an $\mathcal{F}$-process with the name ondoc-rng that performs RELAX NG validation of OMDoc documents according to the OMDoc schema. Then the $\mathcal{F}$-result for the invalid OMDoc document /problems/graphs.omdoc would be:

$\mathcal{P}("omdoc-rng", \{ "doc" \}, \langle \xi(vx:doc("/probs/graphs.omdoc"))\rangle) = \langle false,"Unexpected attribute ..." \rangle$

Here, doc is a parameter name of a document to be validated. We see that the document did not pass the validation and the validation result represents a string error message. If validation were successful, the $\mathcal{F}$-result would be an empty sequence, since RELAX NG does not produce any post-validation result unlike XML Schema validation.

Example 6.2 (Transforming OMDoc to XHTML) OMDoc transformations to XHTML require not only applying XSLT stylesheets but more advanced mathematical notation transformation performed by the JOMDoc [Jom] library (refer to Section 6.5 for a more elaborated example). The process of a particular OMDoc document transformation can be formalized as:

$\mathcal{P}("jomdoc", \{ "doc" \}, \langle \xi(vx:doc("/.../trees.omdoc"))\rangle) = \langle true, <html xmlns="http://www.w3.org/1999/xhtml">...</html> \rangle$

Example 6.3 (XML-Diff) The result of XML-differencing of a theorem from two different branches can be presented as an $\mathcal{F}$-result too:

$\mathcal{P}("xml-diff", \{ "left-element", "right-element" \}, \langle \xi(vx:doc("/.../branch1/trees.omdoc")//theorem[@xmd:id='tree-th']), \xi(vx:doc("/.../branch2/trees.omdoc")//theorem[@xmd:id='tree-th'])\rangle) = \langle true, <vx:diff><vx:insert>...</vx:insert> ...</vx:diff> \rangle$

The $\mathcal{F}$-process model are flexible enough to satisfy many processing workflows utilizing the full power of XQuery (note the usage of the $\xi$ function).

6.2.1 Plugins in TNTBase as $\mathcal{F}$-processes

We have implemented a plugin architecture in TNTBase to allow 3rd-party developers to embed their own $\mathcal{F}$-processes into TNTBase with ease. A number of plugins has been built in the scope of the TNTBase project that are briefly summarized in Section 6.5.

We distinguish three steps of integrating an $\mathcal{F}$-process into TNTBase:
6. ADAPTATION TO THE SPECIFIC DOMAIN

1. Plugin implementation: Defining the logic of obtaining an $F$-result.

2. Plugin configuration: Associating a plugin with a process name to expose the plugin functionality.

3. Plugin Integration: An optional step to integrate a plugin into the pre-/post-committing pipeline.

   The steps above are general enough to be realized in any VXD implementation. Let us consider plugins workflows from their implementation to automation within TNTBase.

6.2.1.1 Plugin Implementation

Each plugin in TNTBase is given access to xAccessor (and therefore to the underlying XQuery processor) and returns an $F$-result. Ability to get arbitrary input parameters as well as utilizing all functionality of xAccessor gives full flexibility to the plugin developer. Essentially, every plugin represents an $F$-process with a not yet defined name. Next section will show the plugin instantiations that represent customizable $F$-processes.

We will not go into technical details of a plugin framework realization in TNTBase since it is purely technical. For an example of the detailed plugin implementation refer to [Rabb].

6.2.1.2 Plugin Configuration

Many software systems employ configuration documents to define auxiliary information about its (sub-)components. In order to do that, we utilize XML files stored in the repository itself in some document called a (Plugin) Configuration Area (CA). In case of TNTBase we used the predefined /admin/processes.xml FS-path for the CA. Other VXD implementations may define a fixed CA location for all repositories, a fixed location per repository, or even allow dynamic modification of the CA location. Having a CA in a repository itself brings a number of benefits:

- A CA file is versioned, that allows, for instance, change tracking, reverting back to the last successful configuration, etc.

- Plugins can be configured remotely by any user that has access to the corresponding repository’s CA.

   Basically a CA allows binding a plugin implementation with names that are unique repository-wide. Additionally, some of the plugin parameters can be predefined.

   As an example, consider a plugin that has two parameters: a RELAX NG schema path with the parameter name schema−location and a document path with the parameter name path. The plugin validates the document against the schema. An example excerpt from a TNTBase CA would look like the following:

   1. We will use the ”plugin” string to represent a “raw” name of an $F$-process induced by the plugin

136
An implementation attribute identifies a plugin realization that performs the RELAX
NG validation. In TNTBase plugins are identified by a Java class name. For the same
plugin implementation we fix two “bindings” that are different in names and also in
predefined parameters that are embedded as child XML elements. In our case the same
RELAX NG validator plugin will get different schema locations, for OMDoc and Doc-
Book. We will call such bindings $\mathcal{F}$-methods to distinguish them from the abstract
notion of $\mathcal{F}$-processes and $\mathcal{F}$-processes induced by plugin implementations.

$\mathcal{F}$-methods are also $\mathcal{F}$-processes. In the example above we see two $\mathcal{F}$-methods that
naturally define two different $\mathcal{F}$-processes ($p$ represents some arbitrary document path):

$$
\mathcal{P}("omdoc-rng", \{"path"\}, (p)) = \\
\mathcal{P}("plugin", \{"schema-location", "path"\}, ("/schemata/omdoc-1.3.rnc", p))
$$

$$
\mathcal{P}("docbook-rng", \{"path"\}, (p)) = \\
\mathcal{P}("plugin", \{"schema-location", "path"\}, ("/schemata/docbook-5.rnc", p))
$$

In TNTBase we extended an xAccessor (and, hence, the TNTBase interface)
to allow execution of $\mathcal{F}$-methods induced by plugins. Parameters not defined in the
CA, must be provided upon $\mathcal{F}$-method execution. For example, validation of OMDoc
and docbook files in TNTBase using $\mathcal{F}$-methods above can be done by a URL of the
following form:

http://tntbase.org/.../method/omdoc-rng?path=/lectures/arrays.omdoc
http://tntbase.org/.../method/docbook-rng?path=/lectures/arrays.docbook

Since in TNTBase we provide a REST interface, the $\mathcal{F}$-result boolean value is
represented via HTTP response codes: in the range 200 – 206 for true, and other
codes for false [Fie+99]. The $\mathcal{F}$-result sequence is serialized and returned in the HTTP
response. Other systems should implement their own mechanisms of communicating
the $\mathcal{F}$-result to an end-user depending on the VXD interface they provide.

### 6.2.1.3 Persisting $\mathcal{F}$-results

We found it extremely useful to allow caching $\mathcal{F}$-results. Since an $\mathcal{F}$-result of every
$\mathcal{F}$-method is an XDM sequence we can utilize an underlying XML-database for storing
$\mathcal{F}$-results. In TNTBase we simplified the model by persisting only the $\mathcal{F}$-results of
$\mathcal{F}$-methods that have a single parameter named path and a single document node as
an $\mathcal{F}$-result sequence. The reason for that is a simpler model of associating $\mathcal{F}$-results
with original documents in TNTBase. In such a way we removed the burden from an
end-user of storing, retrieving and managing persisted $\mathcal{F}$-results.
6. ADAPTATION TO THE SPECIFIC DOMAIN

This is intuitively clear from the following example. Assume that we have an \( F \)-method associated with the name omdoc-\( xhtml \) that takes an OMDoc document path and produces its XHTML representation. If we want to persist XHTML representation we just store an XML document in the database with metadata representing an FS path, and \( F \)-method boolean value and an \( F \)-method name. More precisely, we introduced the following metadata fields for cached \( F \)-results:

- \texttt{tnt:method-name} - an \( F \)-method name
- \texttt{tnt:fpath} - a path for which we cache the \( F \)-result
- \texttt{tnt:success} - an \( F \)-result boolean value

Then retrieving a cached XHTML presentation of the OMDoc document is easy using the following generalized XQuery function:

\[
\text{declare function tnt:f-result($method as xs:string, $path as xs:string, $success as xs:boolean) as document-node() {?}
\]

\[
\text{let $result := fn:collection()[ns:metadata('tnt:method-name', .) = $method] [ns:metadata('tnt:fpath', .) = $path]
if ($success) then
$result[ns:metadata('tnt:success', .) = fn:true()]
else
$result
}
\]

The function above retrieves an \( F \)-result document node if it exists in the database. Additionally, we can provide a parameter whether we want to retrieve all \( F \)-results or only successful ones. We can use the \texttt{tnt:f-result} function to further processing. E.g. we want to retrieve a sequence of all list items within the XHTML representation of the /lectures/trees.OMDoc document:

\[
\]

We demonstrated how persisted \( F \)-results can be accessed and transparently queried in TNTBase. The same approach can be analogously utilized in other VXD systems.

Given that we can cache \( F \)-results only when there is only one path parameter of an \( F \)-method, it becomes straightforward to persist an \( F \)-result when an \( F \)-method is executed. TNTBase REST interface takes care of it \textit{automatically} when an additional cache parameter is provided. For example:

http://tntbase.org/.../method/omdoc-xhtml?path=/lectures/graphs.OMDoc&cache=true

It is worth noting that we do not cover the issue of maintaining persisted up-to-date \( F \)-results. This is intentional as it is a highly technical discussion.

6.2.1.4 Plugin Automatization

Manual processing of documents (i.e. \( F \)-methods execution) plays an important role when, for instance, a user intends to browse a presentation in real time, or validate a
6.2 Towards TNTBase(OMDoc)

particular revision of a collection of documents and assert the validity of data for that revision. In TNTBase \( \mathcal{F} \)-methods are exposed via the REST interface that makes use of the extended xAccessoir and the XQL. It is possible to access a certain \( \mathcal{F} \)-method using its name that serves as an identifier. \( \mathcal{F} \)-method parameters can be encoded in the CA and later on augmented or overridden in the REST request. For example, a user can dynamically provide another RELAX NG schema location to validate other types of documents using the same RELAX NG validator plugin.

However, it is also important to have an ability to automate certain steps so that e.g. validation and XHTML caching is done automatically upon commit. It turned out that it was easy to implement that in TNTBase by utilizing SVN properties. The approach described below is easily adaptable to all VXD systems that have an underlying VCS supporting properties on the file/directory level.

A **Processing Property** (PP) is a custom SVN (VCS) property that describes a number of \( \mathcal{F} \)-methods that will be executed on a document or on a set of documents. We chose the tnt:process property name for this purpose, although other VXD systems, of course, can use different names for it.

In many VCSs, properties are applied directly on files or directories. While it is often possible to set such a property recursively, new added content will not get that property automatically. For us it was very important to maintain “islands” of validity for certain groups of documents. Therefore we had to come up with our own syntax for PP values that would be rich enough to satisfy this requirement.

A PP can be set on files or directories. The Extended Backus-Naur Form (EBNF) for a file PP:

\[
\text{FILE\_PP} = ((\text{METHOD\_NAME}''+''\text{+}'')*, \text{METHOD\_NAME})
\]

Basically, a PP contains one or more method names separated with a plus sign. Order of referenced \( \mathcal{F} \)-methods is not relevant since the order of execution upon commit is defined by the pre-/post-commit hooks considered below. Example of a file PP is ondoc−dtd+omdoc−rng that enforces validation according to the DTD and RELAX NG OMDoc schemata.

A **folder PP** allows to set different \( \mathcal{F} \)-methods for different kinds of files. The proposed EBNF is the following:

\[
\text{FOLDER\_PP} = ((\text{FILE\_PATTERN}, \text{FILE\_PP})*, \text{FILE\_PP})
\]

where FILE_PATTERN is a typical file pattern with wildcards such as paper??.

The semantics is that files with the pattern FILE_PATTERN will be processed using methods described in the adjacent FILE_PP component. If the length of the folder PP list is odd, the last entry is treated as a default method that applies to any file with a pattern that is not in the list of a folder PP.

When processing a file, the entry in its file PP determines the \( \mathcal{F} \)-methods which the file will be a subject to. If there is no matching entry, one is searched in the PP of the parent directories in order until a matching entry is found. If the root directory is reached without finding an entry, the predefined method none is chosen, which does nothing and always succeeds.
6. ADAPTATION TO THE SPECIFIC DOMAIN

Figure 6.2 provides an example of how a PP can be associated with folders and files (PP values are denoted in red). trees.xhtml is not processed at all since it is marked with a predefined F-method none. draft.docbook is validated against the DocBook DTD schema. Other files do not contain PPs, and therefore they are processed according to the PPs of parent folders. For instance, VLDP.docbook will be validated against DocBook RELAX NG schema whereas VLDP.xhtml will not be processed at all (see a PP of the papers folder). tree.omdoc will be validated against the OMDoc RELAX NG schema since the PP of the root folder contains a rule for that and all sub-folders and the file itself does not overrule it.

![Diagram](https://example.com/diagram.png)

**Figure 6.2: An Example of Processing Properties**

We have shown that a user is allowed to define properties recursively unlike the semantics of usual VCS properties – if set on a folder, does not influence its content. The PPs nesting semantics is maintained by the dedicated pre-/post-commit hooks discussed below. A PP set on a deeper file system level overrides a PP set on parent folders. This mechanism allows disabling some types of files in subfolders by setting a PP to none for certain files that should be temporarily excluded from a typical method processing.

A more complicated example of a folder PP set on /materials/ is the following:

```
(*strict*.omdoc advanced−validation+omdoc−xhtml) omdoc−rng+omdoc−xhtml
```

That means, all OMDoc files in the folder /materials/ that contain the word strict will be subjected to the “advanced” validation and XHTML presentation whereas all
other files will be validated using a RELAX NG schema and transformed using the same `omdoc–xhtml` \( \mathcal{F} \)-method.

Depending on the nature of a method it might be desirable to run it right before the final commit to control its success based on some criteria – such as validation –, or right after a commit to perform “side-effect” tasks, such as generation of human-oriented presentation and caching it for later usage. SVN’s commit hook mechanisms were ideally suitable for that purpose – pre-commit hooks for the former use cases, and post-commit hooks for the later use cases. Vast majority of VCSs supports pre-/post-commit hooks as well.

We have separated logic of associating \( \mathcal{F} \)-methods with documents from their execution to give more control and flexibility for automated document processing.

In case of SVN pre-/post-commit hooks are just scripts that are executed right before or after a commit. In our implemented model we have one hook per an \( \mathcal{F} \)-method. All committed files amenable for an \( \mathcal{F} \)-method are aggregated and processed by a single \( \mathcal{F} \)-hook that is generated automatically by our utility script based on the \( \mathcal{F} \)-method name and pre-/post-process semantics. Additionally, an \( \mathcal{F} \)-result can be automatically cached. \( \mathcal{F} \)-hooks make use of the TNTBase REST API for a particular \( \mathcal{F} \)-method (cf. Section \[6.2.1.2\]). A TNTBase administrator is responsible for incorporating \( \mathcal{F} \)-hooks into SVN’s pre-commit or post-commit scripts in the order intended by a desired pipeline. For example, if advanced validation should follow schema validation, then the \( \mathcal{F} \)-hooks should be incorporated in that order.

Thus we can see that the order of method names in PPs does not matter, since it is defined by a TNTBase administrator. Our generated \( \mathcal{F} \)-hooks also take care of aborting a commit if an \( \mathcal{F} \)-result boolean value was negative for a particular \( \mathcal{F} \)-method. It holds only for pre-commit \( \mathcal{F} \)-hooks. In case of a post-commit \( \mathcal{F} \)-hook only a warning is returned to a client in case of an error.

However, the hook mechanism is only one way to achieve interoperability between components and supply VXD(\( \mathcal{F} \)) with automated document processing. Another option could be a tighter integration of \( \mathcal{F} \)-method within an underlying VCS. But the hook approach naturally fits into a VCS architecture and considerably diminishes designing and implementation efforts.

It is also important to note that due to the full XQuery support in the plugin interface, \( \mathcal{F} \)-methods can interoperate with each other by accessing cached \( \mathcal{F} \)-result information from another \( \mathcal{F} \)-method. For instance, XHTML generation may embed some RDFa annotation from RDF content generated by another \( \mathcal{F} \)-method (a real life example of that is provided in Section \[6.5\]).

### 6.3 XML-Diff as a TNTBase Plugin Example

In order to summarize the discussed approach let us provide an example of a TNTBase plugin integration that would not be that easily achievable without having a solid underlying implementation of the VXD concept. Section \[2.5.2\] explained why we do not have XML-diff algorithms as a realization of diff/patch functions (Definition \[2.6\]).
6. ADAPTATION TO THE SPECIFIC DOMAIN

Appendix B.7 discusses further why XML-diff is not integrated into the core of the VXD concept.

Assume that for every committed OMDoc document we want to cache the XML diff between the HEAD revision and its prevision revision. Moreover, we want to forbid a commit of OMDoc files that do not contain any theorems. We need to undertake the following steps, the most important of which is a plugin implementation.

Plugin Implementation We need a plugin implementation that receives a document path and checks whether the HEAD version contains a theorem. If it does, we want to generate an XML diff. The XQuery pseudo-code for the plugin is the following:\footnote{In TNTBase plugins are implemented in Java. However, for the sake of clarity we utilize the pseudo-code syntax that we have seen before in this work}

\begin{verbatim}
(: Implementation of the plugin interface :) 
declare function tnt:diff−plugin($path as xs:string) { 
  let $doc := tnt:doc($path) 
  if (fn:empty($doc)) then 
    (fn:false(), "Document does not exist or not XML") 
  else 
    if (fn:empty($doc//omdoc:theorem)) then 
      (fn:false(), "Document does not contain theorems") 
    else 
      (: Getting the HEAD revision numeric value :) 
      let $latest−rev−num as xs:integer := tnt:latest−rev−num(); 
      (: Getting a previous revision of a document :) 
      let $previous−doc := tnt:doc($path, $latest−rev−num − 1) 
      if (fn:empty($previous−doc)) then 
        (: A document has just been added => success with no XML−diff :) 
        (fn:true(), ()) 
      else 
        (fn:true(), tnt:xml−diff($doc, $previous−doc)) 
  };
\end{verbatim}

We assume that we have the function tnt:xml−diff that performs XML differencing of two XML nodes. Also we have two TNTBase external XQuery functions in the XQL to access previous revisions of XML documents and getting the latest repository revision number. Note that the first element of returned sequences are boolean values.

Plugin Configuration Now we have to assign a name in the CA for the plugin above. We do not need any pre-defined parameters, so the body of the process element is empty.

\begin{verbatim}
<processes>
  ...
  <process name="tnt−diff" implementation=org.tntbase.TntDiffPluginImplementation/>
  ...
\end{verbatim}
**Setting a PP**  Assume that we want to process OMDoc documents located only in the /trunk folder: We have to execute the following SVN command assuming that we are in the working copy root:

```
svn propset tnt:process "(*.omdoc tnt−diff)" trunk
svn ci −m"Adding a tnt:process property for OMDoc diffing"
```

All OMDoc files in the trunk folder will be a subject of the tnt−diff F-method unless no other PP overrides this rule.

**Generating and Installing the F-hook**  We make use of the TNTBase hook generator script:

```
> python hook−generator.py −−method=tnt−diff −−cache−result=yes > tnt−diff−hook.sh
```

We provide a method name and request caching of F-results − TNTBase will automatically take care of caching.

The very last step is to integrate tnt−diff−hook.sh into the SVN’s pre−commit script − the generated hook script will take care automatically of aborting a commit if some OMDoc file does not contain any theorems.

**Accessing Cached Results**  TNTBase will automatically generate XML−diffs between consecutive revisions of OMDoc documents in the trunk. Since a plugin requires only one parameter with a name path, caching of F-results is supported in TNTBase out of the box. If we committed a new version of /trunk/graphs.omdoc we can easily figure out a number of changed XML nodes using XQuery:

```
let $diff := tnt:f−result("tnt−diff", "/trunk/graphs.omdoc", true)
return count($diff/element())
```

As we saw from a workflow definition above, the main challenge for any format−specific task is to implement the logic of an F-process itself and our VXD implementation manages the rest automatically. The TNTBase plugin interface is general enough to cover wide range of use cases (see Section 6.5 for more examples) and the access to the XQuery processor allows plugins to make use of information in e.g. VDocs or content cached by other plugins.

### 6.4 Related Work - XProc

The most prominent related work was performed by the W3C Consortium that developed XProc [WMT10] − a language for XML pipeline processing. XProc is a mature specification that has its obvious advantages:
6. ADAPTATION TO THE SPECIFIC DOMAIN

- XProc is a W3C recommendation which practically means better elaboration and more advanced support from multiple vendors (e.g. xDB or eXist).

- Order of processing tasks is fixed and clearer whereas in TNTBase the order of execution is decoupled from F-method semantics.

- In TNTBase interoperability is maintained via persisting F-results in the database which is not desirable in all cases and may negatively affect performance.

- Many other XML technologies are typically supported out of the box in XProc engines, e.g. XSLT or XInclude [MOV06].

On the other hand, the TNTBase(F) approach has its own advantages:

- XProc does not have pre-/post-commit semantics which is important for repository-like data stores.

- The TNTBase approach has a tight integration with XQuery and the underlying XML database.

- TNTBase automatically takes care of persisting intermediate or final results of the steps in the underlying XML database. These data can be processed out of the pipeline scope.

- The approach can be seamlessly applied for folders or VDocs as well.

- The approach is integrated with a versioning model. VCS properties become a natural means to flexibly control F-processes.

6.5 Examples of Implemented Plugins

In course of the development TNTBase gained a number of plugins that are described in this section. Some of them proved to be highly useful for the OMDoc domain and therefore they are shipped together with a default TNTBase installation. The multitude of plugins being used can be considered as an evaluation metric for the usability and applicability of the F-processes approach and TNTBase in general.

The following plugins are being used the most within different TNTBase installations:

**JOMDoc** The JOMDoc project [Jom] is a rendering framework for MathML and OMDoc documents. Since generating of HTML5 or XHTML OMDoc presentation was highly important for us, JOMDoc has been integrated into TNTBase as a plugin. OMDoc presentation is not cached and generated automatically on each request.
Krextor The Krextor project [Lan09] is aimed at extracting RDF data from XML documents. Originally it was used to extract RDF from OMDoc documents. Since it is very practical to store RDF data together with its OMDoc sources, Krextor was integrated as plugin into TNTBASE. Cached RDF is used by the JOMDoc plugin in order to embed RDF data into XHTML representation of OMDoc documents that allowed us to integrate RDFa-based semantic services into XHTML. This is an example of how F-results of different plugins are used in a pipeline.

Mocassin Mocassin [Zhi] is a project for searching mathematical facts based on RDF information contained in the Virtuoso [Olv] triple-store. In order to expose Mocassin features to collection of OMDoc documents stored in TNTBASE, a plugin was developed that takes information extracted by Krextor and propagates it to the Virtuoso instance. This is an example of a plugin that also makes use of another plugin F-results and performs some “side-effect” jobs by persisting information to another storage.

X-Diff One of the built-in XML-diff plugins integrated into TNTBASE is based on the X-Diff [WDC02] library. It can be used directly from XQuery for XML element differencing and provides a syntax for XML differences that is more user-friendly than the one utilized for enabling VDocs editing.

Latin The Latin project [KMR] employs TNTBASE as its underlying storage system. It also makes use of TNTBASE F-result caching to persist RDF-like theory (ABox) documents. These documents are later utilized for stricter consistency checks (e.g. checking for infinite loops, undefined definitions, etc.) performed by another validation step.

6.6 Conclusion

We have presented a concrete implementation of the VXD(F) concept (as TNTBASE(F)) and exemplified it on the OMDoc-related applications. We had to spend only a tiny amount of efforts of adapting TNTBASE to TNTBASE(OMDoc) in comparison to the core TNTBASE implementation. This demonstrates that the VXD toolkit is indeed practical, and all of the presented theoretical concepts truly contribute to the areas that deal with XML.

We have given a general overview on how VXD developers can adapt their XML formats using techniques presented in this chapter. Introducing an F-specific layer in a VXD considerably reduces the application logic of (web) applications that manage large, changing XML-based collections of documents.

We envision even richer applications of TNTBASE by utilizing its format-specific functionalities. Let us look at the presentation of an ontology in Figure 6.3. This example from [Lan11] shows a heavily cross-referenced XHTML +MathML rendering of the underlying OMDoc sources, where much of the structural relations have been annotated.
6. ADAPTATION TO THE SPECIFIC DOMAIN

Friend of a Friend (FOAF) vocabulary

imports from: wordnet, de, owl, quant1, logic1

AXIOM: The foaf:Person class is a sub-class of the foaf:Agent class, since all people are considered

Person ⊑ Agent

AXIOM: Person ∩ Organization = ⊥

CONCEPT: made

The foaf:made property relates a foaf:Person to a foaf:Agent, with a relation made = maker^-

TYPE: ObjectProperty(Agent, Thing)

AXIOM: made = maker^-

LEMMA: maker = made^-

PROOF:
1. We know that made = m
2. Interpreting the above (well-formed) semantic, this means that
   made^1 = (maker^-)^1 = (maker^-)
3. Now we apply the inverse on both sides, eliminate double inverses, and obtain
   (made^1)^- = (maker^-) = maker^-
4. This is just the interpretation of maker = made^-, which we had to prove.

CONCEPT: membershipClass

The foaf:membershipClass property relates a foaf:Group to an RDF class representing a sub-class of

foaf:Agent whose instances are all the agents that are a foaf:member of the foaf:Group. See foaf:Group for

details and examples.

AXIOM: ∀ m, g, C. (g ∈ member m ∩ membershipClass(g, C) ⇒ m ⊑ type C)

Export OWL as RDF/QL

Figure 6.3: A complex presentation of a documented Ontology

in RDFa. The rendering and RDFa annotation process made use of the extracted interface
information, which is then picked up by JOBAD [GLR09], a JavaScript library
that instruments the XHTML for interactivity. For instance, JOBAD can use CSS
properties to collapse a proof or embedded MathML maction elements or pick a differ-
ent (pre-generated) notation variant. But JOBAD can also use AJAX-style callbacks to
the TNTBase(OMDoc) system for definition lookup: For any symbol a right-click menu
item will query TNTBase for the definition of the corresponding constant, which will
then be served by TNTBase(OMDoc) and displayed in a popup by JOBAD. Pre-existing
validation, annotation, rendering, and interaction functionalities were integrated into
TNTBase(F), which also serves as a web application platform [Koh+09]. The work of
extending TNTBase was influenced by discussions of how to integrate TNTBase into
the systems of the KWARC research group, e.g. in the SWiM system. The latter is
an OMDoc-based semantic Wiki for scientific/technical documents [Lan08; Lan11], in
which workflows similar to the ones described in this chapter had to be established for
the integration of the OMDoc format into the underlying IkeWiki platform.

We are using TNTBase(OMDoc) in daily practice exploring the possibilities of
TNTBase(F) for a semantic document format that is designed to support document
aggregation and to stretch the limits of document modularization. We are also looking
into the use of TNTBase(OWL) for ontologies, which is semantically structured docu-
ment format, but does not (directly) support document aggregation with the intention
of developing a suitable generic XML extension that generalizes XInclude [MOV06] to
queries.
Future work would be to experiment more with “management of change” in TNT-Base(\mathcal{F}): If the dependencies between document fragments are made explicit (e.g., by generating them during the interface extraction), the database can compute what other documents are affected by a change. If validation or presentation generation is enabled, they can be revalidated or rerendered automatically. The affected documents do not have to be known to the author of the changed documents.
6. ADAPTATION TO THE SPECIFIC DOMAIN
Chapter 7

Case Studies

XML is like violence – if it doesn’t solve your problems, you are not using enough of it.

Anonymous

7.1 Introduction

This chapter presents the major applications of TNTBase as a document management system/repository and discusses those features that were particularly beneficial. Previous parts mostly discussed theoretical concepts and abstract engineering techniques whereas this chapter shows the practical value of the TNTBase system. Sections 4.6 and 6.5 already showed the practical benefits of virtual documents and the plugin architecture, but this chapter will provide a wider picture of the world where TNTBase is used. The author considers TNTBase use cases as the main and the most important evaluation approach since TNTBase is a unique system of that kind (best to the author’s knowledge).

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Versioning (Ch. 2)</th>
<th>Querying (Ch. 3)</th>
<th>F-methods (Ch. 6)</th>
<th>Plugins (Ch. 4)</th>
<th>VDocs (Ch. 1)</th>
<th>Tag Libs (Ch. 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenCS Corpus</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Planetary</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>DocTIP</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>BioPortal</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Linked Data</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Logic Atlas</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>C&amp;L</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 7.1: TNTBase Features per Use Case
7. CASE STUDIES

We will start with separating the classes of the most notable TNTBase features. The main part of the chapter elaborates the major case studies. Section 7.7 concludes the chapter by giving some insights of possible TNTBase applications in the future.

Table 7.1 provides a brief summary of what TNTBase feature classes are used in what case studies. The most notable features of TNTBase are support for document versioning, XML querying facilities, an infrastructure for user-defined XML validation and presentational workflows and an implementation of the virtual documents concept. Tag libraries are not yet used in TNTBase since it is a young technology that has been developed only recently.

Case studies are diverse and each of them deserves a separate section (Sections 7.2 – 7.6). We have mostly experimented with General Computer Science lecture notes documents, the math-enabled informational portal Planetary [Koh+11; Dav+; Pla], a change management system DocTIP [Doc] and the ontology repository BioPortal [BO09]. We also use TNTBase for publishing mathematical Linked Datasets and storing modularized math theories in the scope of the LATIN [KMR] project. As an industrial example, we are employing TNTBase as a storage and processing tool for the publishing company C&L homepage.

7.2 General Computer Science Lecture Notes at Jacobs University

7.2.1 TNTBase – A Repository for OMDoc

Our research group has accumulated a collection of more than 4000 lecture note slides and homework/exam problems. They are originally written in sTeX [Ste], a LaTeX-based input syntax for OMDoc. Besides being compiled to PDF for presenting and reading, they can be translated to OMDoc using the \LaTeX\ → XML converter [Mil]. The JOMDoc library [Jom] integrated as a plugin can render the OMDoc semantic markup to human-readable XHTML combined with MathML for presenting mathematical formulae [Aus+10]. These steps had been performed semi-automatically in the time before TNTBase: it was a setup of Makefiles that were invoked manually to produce XHTML content. Beyond that, we wanted to answer queries such as “what exercises deal with structural induction?” (Figure 7.1 shows an example of retrieving the most complicated problem about graphs and trees), generate exams automatically (e.g. by selecting exercises that cover certain topics, have not been used last year, and sum up to one hour), and to make the lecture notes browsable interactively (e.g. by showing the definition of a mathematical symbol in a popup, when a user clicked on an occurrence of it in a formula). These features were enabled by maintaining the lecture notes in TNTBase and integrating plugins for certain tasks.

Additionally, we made use of the VDoc functionality by implementing VDoc Specs that enable useful views on the GenCS corpus:

[1] Currently, JOMDoc also supports HTML5 [Hic10]

[2] Interactivity is supported by the JOBAD framework [Koh+09]
7.2 General Computer Science Lecture Notes at Jacobs University

Figure 7.1: The TNTBase Web-interface for the GenCS Corpus

- A VDoc for automated exam generation that was discussed in Section 4.6.1. Materialized versions are used for publishing a final version of the exams. Minor technical issues need to be solved in order to use them reliably in the GenCS course for real examinations.

- A VDoc for the generation of theory dependency graphs in the GenCS corpus (Section 4.6.5). Graphs are represented in the DOT language [GKN06] which is further rendered to SVG for human-readable presentation by means of $\mathcal{F}$-processes implementation (cf. Chapter 6). We found it useful for visualizing theory dependencies to help getting more insights on the corpus structure and help developers browsing through OMDoc documents or their XHTML presentations.

- Since OMDoc documents contain information about conceptual dependencies (e.g., theory imports), we were able to create a VDoc for generating guided tours [MU01], i.e., self-contained sub-documents that introduce the necessary prerequisites of a particular concept. As VDocs allow the re-use of parametric XQuery statements that operationalize e.g., the topological sorting of concept...
7. CASE STUDIES

descriptions, populating a file system with guided tours over a content collection becomes a mechanical exercise. In fact, we conjecture that much of the functionality of advanced e-learning systems such as ActiveMath [Mel+03] can be externalized into VDocs.

One of the most important improvements that still is not made is to create a \LaTeXXML plugin that would be triggered after committing an sTeX document and would convert it automatically to OMDoc. These OMDoc documents can then be indexed and queried, or be checked for validity according to the OMDoc RELAX NG schema. The latter ensures that sTeX sources and the \LaTeXXML converter perform in a desired way.

In conclusion, the GenCS corpus provides a solid playground for nearly all TNTBase features that are all used on a daily basis. The next section will showcase the system that connects end-users and TNTBase by adding graphical user interface and document interactivity.

7.2.2 Planetary – A Math-Enabled Information Portal

The Planetary system (see [Koh+11; Dav+] for an introduction) is a math-enabled Web 3.0 information portal which combines the social, user generated Web (Web 2.0) with semantic features. The Planetary system can be instantiated by a user community to individual instances that aggregate material on a specific topic of joint interest. One of the most relevant instantiations for us is the one for the GenCS corpus [Koha] introduced in the previous section. In a nutshell, Planetary supplies a user with a powerful web-interface that serves as a frontend for the data stored in the GenCS TNTBase repository. This emphasizes that TNTBase can be used not only as a standalone application, but also as a component of other software systems.

Authors are able to create sTeX documents that are automatically converted to OMDoc sources and committed to TNTBase that indexes them. Then the Planetary system can perform server-side semantic services via TNTBase utilizing user-defined XQuery queries. TNTBase thus becomes a source of the user-adaptive, custom-generated documents forming the Planetary content commons. Many semantic services can directly be derived from this setup, mainly two services are provided by TNTBase:

i) Definition Lookup: All technical terms and symbols in formulae presented in Planetary are linked to their semantic counterparts in the content commons, which in turn are linked to their definitions [GLR09].

ii) Prerequisites Navigation: As the content commons has an inherent notion of semantic dependency, we can use that to show prerequisites leading to a concept. Currently TNTBase provides two ways of dealing with prerequisites: i) a concept graph view, where the required concepts can be navigated on demand by clicking on concept nodes, and ii) guided tours, where the necessary content is generated in a coherent narrative [Müll10a].
Last but not least, TNTBase serves as a mediator between Planetary and the management of change system DocTIP. This is a subject of the next section.

### 7.2.3 DocTIP – Management of Change via OMDoc

The DocTIP system \[\text{Doc}\] provides a generic framework that combines sophisticated structuring mechanisms for heterogenous formal and semi-formal documents with an appropriate change management to maintain structured relations between different documents. It is based on abstract document models and abstract document ontologies that need to be instantiated for specific document formats, such as OMDoc. The heart of the system is the document broker, which maintains all documents and provides a generic update and patch-based synchronisation protocol between the maintained documents and the connected components working on these documents. Components can be authoring systems, or analysis and reasoning systems offering automatic background processing support, or simply a connection to a repository allowing to commit and update the documents.

When the document broker obtains a change for some of its documents, the changes are propagated to all connected components for that document. A configurable impact
analysis policy allows the system designer to define if impact analysis is required after obtaining a change from some component. To perform the impact analysis the document broker uses the GMoC (General Management of Change) tool [Hut04] to compute the effect of the change on all documents maintained by the document broker. The GMoC tool returns that information as impact annotations to each individual document, which are subsequently distributed to all connected components by the document broker.

In order to add change management support for the OMDoc authoring workflows, we consider the informational architecture of multiple systems: Planetary, TNTBASE and DocTIP (see Fig. 7.3). To recapitulate the important parts here from the previous section, a user interacts with the Planetary system via a web browser, which presents the mathematical knowledge items based on their XHTML+MathML presentation in a Wiki-like form. The XHTML+MathML documents are rendered from content oriented mathematical knowledge items in OMDoc format. Along with the XHTML+MathML document versions, the Planetary system maintains the original sTeX document snippets, which the author can edit in the web browser. The OMDoc documents are maintained in the TNTBASE repository together with their original sTeX source snippets. Any change in the OMDoc documents in TNTBASE results in an update of the corresponding knowledge items in the Planetary system after rendering the OMDoc in XHTML+MathML. Upon edit of the sTeX snippets in the Planetary system, a new OMDoc document is created from the sTeX sources [GSK11] and pushed into TNTBASE, which returns the XHTML+MathML presentation.

To add change management support, we connected the DocTIP-system to TNTBASE which returns impact information in the form of OMDoc document annotations, which are cached in the TNTBASE in the XML form (recall $\mathcal{F}$-results caching discussed in Section 6.2.1.3). Those impacts can be retrieved at any time, or queried and aggregated using XQuery.

Like modifications to the OMDoc file, changes in the impacts cache are pulled from TNTBASE by the Planetary system. The rendering of OMDoc in XHTML+MathML preserves the xml:id. Therefore, the Planetary assigns the impacts to the rendered snippets using the for-attributes and presents the document on the Wiki-page. Thus a user edits sTeX source but gets impacts in the form of the highlighted parts of the XHTML+MathML presentation (Figure 7.5). Such transparent interaction should simplify the process of analyzing possibly incorrect changes and facilitates semantic

---

1 The figure is taken from the joint work [Aut+11]
2 The whole setup relies on sTeX back-pointers in the form of \texttt{srcref}-attributes.
7.3 Publishing Linked Data Using TNTBase

The combination of three systems (Planetary, TNTBase and DocTIP) uses the semantic relations that were integrated into the documents to improve their interactivity and make it possible to propagate change impacts to the other documents. That ultimately helps authors to keep the collections of original documents consistent without sacrificing the convenience of editing. For the full story refer to \cite{Aut+11}.

7.3 Publishing Linked Data Using TNTBase

We have experimented with publishing General Computer Science lecture notes (Section 7.2) as Linked Data \cite{HB11}. TNTBase is used as a storage backend as well as an engine that is responsible for supporting content negotiation as well as supplying a user with a requested document format.

The Krextor RDF extraction plugin (cf. \cite{Lan09}), which is run on every commit, extracts structural outlines and metadata from a committed version of an OMDoc document into an RDF representation in order to (i) provide semantic querying possibilities beyond XQuery\footnote{The advantage of querying RDF (in the SPARQL \cite{HS10} language) is that an RDF representation can abstract from different structural dimensions of knowledge being represented in syntactically different ways in XML.} (ii) to be able to enrich the rendered documents with semantic

![Figure 7.4: A Planetary Impact Resolution Dialog](image)
7. CASE STUDIES

annotations, to which interactive services can attach (e.g. in-place lookup of relevant linked information), (iii) and to publish the contents of a repository as Linked Data, so that external semantic search engines and mashups can make use of them [Dav+10]. In order to store RDF scalably, to answer SPARQL [HS10] queries, and to publish RDF as Linked Data, we put the RDF extracted by Krextor into the Virtuoso RDF database (“triple store”) [Olv], which is connected to TNTBase by another plugin.

Storing RDF data in Virtuoso made it possible to use the Mathematical Semantic Search engine (Mocassin) [Zhi] for semantic mathematical search over the GenCS corpus (Figure 7.5). Finally, there is a plugin that integrates the JOMDoc library, which renders OMDoc human-readable XHTML+MathML documents and now also enriches them with semantic RDFa annotations [Dav+10]. Currently TNTBase serves 3 document formats according to the Linked Data principles[^1]: OMDoc source documents, XHTML+MathML presentation and RDF. It would be straightforward to augment the “Linked Data module” of TNTBase so that it serves the original sTeX sources as well.

![Mocassin Web-interface](image)

**Figure 7.5:** Mocassin Web-interface

7.4 BioPortal – An Ontology Repository

BioPortal [BO09] is an open-source repository for OWL ontologies. Initially it was developed for biomedical ontologies, but it was forked by a research group at University of Bremen. Use cases of that research group required sophisticated ontology versioning. The original version of BioPortal uses an extremely primitive versioning model based on the file system: Different versions of ontologies are stored as fulltext in files on the system.

[^1]: Content negotiation is not strictly a Linked Data principle, however, it is often done together in practice.
BioPortal – An Ontology Repository

This approach has a number of obvious disadvantages:

- Storing fulltext for text-based formats are not space efficient.
- There is no easy way to “check out” all ontologies or a subset of them from BioPortal.
- There is no notion of a BioPortal repository revision, and therefore it is impossible to get a grip on history of changes.
- There is no mechanism for collaborative offline editing (working copies) like most VCSs offer.

Integrating TNTBase as an underlying storage for ontology documents allowed us to eliminate all the drawbacks described above. Every BioPortal installation utilizes an underlying TNTBase repository as a storage backend. Users are able to check out the whole repository or a part of it at once using any SVN client thus benefiting from the features that SVN offers, i.e. support update/merge/commit workflows. That allowed ontology developers to work on the same ontology simultaneously committing their changes back upon necessity. However, an SVN client is not the only way of adding/changing ontologies in BioPortal. Users can access ontologies:

1. **Via BioPortal GUI/REST interface** An initial version of an ontology must be committed via this method in order to populate basic ontology information such as creation date, name, emails addresses, etc. This would be more complicated via an SVN client, although would be theoretically possible by making use of dedicated SVN properties for describing metadata. Revisions of existing ontologies can be committed via this method or the next method.

2. **Via SVN client** Once an ontology is in the repository, further modifications are allowed via an SVN client. On every commit the custom pre-commit hook notifies BioPortal so that it can update and synchronize its internal state with the latest TNTBase repository version. If BioPortal fails to update its state, then the pre-commit hook returns an error and the committing transaction is aborted. This mechanism guarantees integrity of data. This is an example of yet another TNTBase plugin that communicates with a 3rd-party system and is initiated via the SVN hook mechanism.

TNTBase also supports committing ontologies in non-XML format. In this case, however, XML-related features of TNTBase, obviously, will not be available. Having ontologies in the XML format brings certain advantages for the use case:

- XML-diff (Some insights can be found in Section 6.3) between different ontologies or revisions of the same one. TNTBase has a number of built-in XML differs

---

1Here we consider XML as a text-based format too since it is typically stored as uncompressed plain text.
which produce raw XML-diff output in a certain syntax. However, an own differ which takes into account the ontology structure can be implemented either in XQuery or as an external plugin.

- XQuery functionality for the whole content of the repository was needed. An extension of the BioPortal REST interface has been developed to communicate requests to the underlying TNTBASE repository.

- Schema validation of ontologies in XML format was required. That was achieved by utilizing the TNTBASE plugin framework and its built-in plugin for XML schema validation.

Deletion of an ontology from BioPortal leads to removing of this ontology from the HEAD revision of the TNTBASE repository, but the history of previous revisions is preserved and can be accessed via an SVN client.

7.5 Logic Atlas

In the LATIN project (Logic Atlas and INtegrator), we deal with more than 300 highly modularized and interlinked logical theories that have been formalized in OMDoc. Similarly to the lecture notes scenario (Section 7.2), these documents have to be rendered to human-readable presentations. This is realized by the MMT [Rab09; Raba] (Module system for Mathematical Theories) web server application, which obtains the formalized documents as well as instructions for how to display formal mathematical symbols from different locations in an underlying TNTBASE repository. Additionally, validation of the mathematical theories was required – a structural validation that extends beyond XML schema validation. For that purpose, the MMT library has been integrated into TNTBASE as another validation plugin [Rabb]. After schema validation, it extracts the formal logical structure of an OMDoc document to an RDF-like representation representing, for example, whether one mathematical symbol is defined in terms of other symbols, whether other symbols occur in its type declaration, or whether a mathematical theory imports other theories. Based on that information, it checks for logical validity. If, for example, a definition uses symbols that have not been imported before, this definition and thus the whole document is not valid. We refer to [KRZ10] for further details.

7.6 Publisher’s Homepage Generation

C&L (Computer- und Literaturverlag GmbH [Cul]) is a small German computer book and magazine publisher. In March 2011 they were looking for a new way of keeping their homepage up to date, consistent, and valid, with little effort per update. So far, they had been using the Perl-based WML toolkit (Website Meta Language [Eng]), which allows for generating HTML in a “make”-like process from source files that internmix
HTML snippets with macros and Perl program code. WML, however, had no longer been maintained since 2006 and no longer worked with the recent Perl 5.12.

C&L’s requirements for a new homepage generation tool were as follows, roughly ordered by decreasing importance:

1. It should be possible to automate the whole homepage generation process. Unless any change to the page structure or layout should be necessary, it should be sufficient to upload the files related to a new publication (i.e. the table of contents, a PDF extract, and the cover image), plus the metadata (including title, author, price, ISBN, number of pages, etc.), whereas the following pages should be (re)generated automatically: the page about the publication itself, as well as links to that page from the main page, from the RSS newsfeed, from a listing of new publications, from a listing of all publications on the same topic (e.g. “books about operating systems”), and from the ordering form.

2. To ensure consistency and easy maintainability, only one document template per page type should be necessary, e.g. one template for a book page, and one for a listing of all books about a certain topic.

3. Given the time constraints of the webmaster, it should be possible to develop these templates incrementally, starting from the old WML templates, or from the static (X)HTML files.

4. No software installation should be required on the server; the webmaster would just upload static (X)HTML files via FTP after (re)generating them. Except for some embedded JavaScript and a few server-side PHP scripts (for processing orders, and for searching the homepage), the homepage consists of static files.

5. The homepage sources should be archived in a versioned repository, so that earlier versions could be retrieved easily. (This had never been the case before, but was generally considered desirable.)

The webmaster chose TNTBase, set up one TNTBase repository for the homepage, and achieved the requirements as follows:

1. The data files related to each publication were added as files to the repository. The table of contents of a book was encoded in XML for easier querying. The metadata of all publications were entered into a single XML file following an ad hoc schema.

2. The document templates were created as VDoc Specs and then instantiated multiple times. For example, the “book page” template is instantiated once per book, passing the ID of the book as a parameter. Using this ID, the queries employed by this VDoc obtain further information about the respective book from the file repository (where names of the RDF and image files start with the book ID) and from the database XML file (where the book is represented by an element with an according XML ID).
3. Incremental migration from the existing static (X)HTML files was possible in that the skeleton of a VDoc Spec can have entirely static content, which one can incrementally replace by queries.

4. Installing TNTBase on the C&L server would have saved the webmaster some work in regenerating files. However, it was also possible to automatically retrieve the (X)HTML files from the TNTBase installation hosted in a different place by a shell script that accesses TNTBase’s REST API with a command-line HTTP client, and then uploads the static files via FTP.

5. TNTBase’s SVN integration satisfied the archiving requirement.

7.7 Future Plans for Using TNTBase

**Mizar Library**  JOSEF URBAN intended to store his Mizar [UB06] library documents in XML format [Urb06] in TNTBase. There was about 10 GB of XML data. Indexing and querying facilities for XML was needed to support his Mizar Wiki [Urb+10]. TNTBase would suit this use case well. Apart from querying and indexing capabilities, validation and presentational workflows were required. Unfortunately, the latest release of the underlying DB XML has a bug¹ that causes significant degradation in database write performance when more than a certain amount of indices (more than 10 in our experiments) are present in the database.

**Ontology Metrics in BioPortal**  As discussed in Section 7.4 TNTBase serves as a backend for the forked BioPortal version and provides additional query facilities for ontologies represented in XML. Whereas queries can be arbitrarily diverse and complex, one of the most desired use cases is computation of ontology metrics: e.g. the number of classes in the whole repository, the number of concepts that have annotations, etc. Currently this information is processed in the background which takes considerable amount of time. Querying facilities together with TNTBase indices are expected to improve overall performance. A drawback of using XQuery for metrics computations is that ontologies in different syntaxes (e.g. RDF/XML [Bec04] or OWL 2 XML [MPPS09]) require different XQuery queries for obtaining information of the same kind. However, once metrics queries are developed for a particular syntax, they can be bundled with a BioPortal installation. A short discussion on this topic can be found in [LZ10].

**Maps of Bremen**  MIHAI CODESCU from DFKI [Dfk] (Deutsches Forschungszentrum für Künstliche Intelligenz²) intended to store Bremen maps [Cod+11] in XML format in TNTBase in order to leverage efficient querying. The total size of the maps was

¹The bug is not exposed online, but has been confirmed in personal communication with GEORGE FEINBERG from ORACLE.
²German Research Center for Artificial Intelligence
20 GB and several files were more than 1 GB themselves. This use case revealed a drawback in the TNTBase implementation induced by metadata storage mechanisms in DB XML. TNTBase stores with each document a metadata field that represents the same document for fast retrieval and precise white-space preservation. Whereas putting an XML document into DB XML can be done in a streaming fashion, the same cannot be done for metadata fields, that is a metadata field should be in main memory before persisting it in DB XML. Since some of the map documents are extremely large, they do not fit into main memory causing an error. This problem can be overcome by modifying DB XML so that it supports metadata streaming. Oracle developers have this feature on their agenda.

7.8 Conclusion

This chapter presented the major case studies we have conducted with the TNTBase system and explains why certain features are particularly useful. As we have seen all features except tag libraries (see Table 7.1) are widely used. The tag libraries approach is still quite young but we believe has lots of potential. Michael Kohlhase is planning to experiment with it for generating presentation from OMDoc files by partly substituting functionality of existing XSLTs. Although the use cases are mainly concerned the OMDoc field, we also saw that TNTBase can be used in other domains as a standalone XML database or as a sub-component of other software systems. Case studies of e.g. BioPortal and the publishing company proved that in practice. That allows us to claim with a high degree of confidence that TNTBase can be applied in many other domains as an XML-enabled repository as well as format-aware versioned XML database. Virtual documents – an implementation of the XML database views concept - may also be an additional convincing factor for some developers.

The practical value of TNTBase justifies the theoretical work done in the scope of this thesis. The next chapter will show that TNTBase provides not only a rich set of features and high usability, but at the same time is indeed a performant versioned XML database.

---

1 We did not consider splitting those files
2 The conclusion is made based on personal communication with George Feinberg
7. CASE STUDIES
Chapter 8

Performance and Stability

As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications.

Dave Parnas

8.1 Introduction

The important aspect of all engineering techniques and methods is the ability to efficiently integrate them into the real world scenarios and the opportunity to use the proposed enhancements with benefit. One of the strongest promises of the VXD concept is the ability to work with any of its implementations transparently if it was a version control systems or an XML database. TNTBase is a successful implementation of the VXD concept, and therefore can be considered as a full-featured SVN and Berkeley DB XML. In our implementation we do not sacrifice practicability and usability in comparison to standalone SVN and DB XML. We will deliberately not compare TNTBase to other XML databases or VCSs since in general performance of VXD’s underlying VCS and XML DB directly influences the performance of the VXD implementation itself. Additionally, best to the author’s knowledge, TNTBase is the only system of its own kind, and therefore we cannot compare it to analogous systems.
8. PERFORMANCE AND STABILITY

We will also not perform an extensive benchmarking of TNTBase since the use cases in the previous section showed that our software can be integrated in the real workflows with ease and success and cope with decent amount of data. However, we have made some basic performance tests to demonstrate that TNTBase is much more than just a prototype or a proof of concept.

In previous sections we saw that TNTBase is not less functional for an end-user than SVN and DB XML at the same time. In this section we will start with performance tests that demonstrate that TNTBase is not much different from the efficiency prospective either. For the tests, the same workstation with an unchanged configuration has been utilized.

8.2 Comparing to SVN

Two most frequently used commands in any VCS are check out/update and commit commands (or their analogues) that correspond to the read/write operations, correspondingly. We will compare performance of these operations for xSVN and unmodified SVN. Tests have been run against xSVN based on SVN 1.4 and unmodified SVN 1.6.

All tests have been performed locally eliminating noise of network latency and throughput. We utilized two data sets, a small dataset A and a bigger data set B. These are subsets of General Computer Science lecture notes at Jacobs University. The files in question represent mixed content, i.e. OMDoc XML documents together with sTeX sources and other binary formats such as PDF, images, log files, etc. XML files together with non-XML files represent real life scenarios that we were constantly dealing with.

The smaller subset contains about 5 MB of data among which there are 34 OMDoc files. The bigger test subset contains 571 MB of data with 1670 OMDoc files. Additionally we had a TNTBase repository that was created with OMDoc-specific XML indices in order to test their impact on the write performance.

Table 8.1 represents the average run times for test case A. In turn, Table 8.2 represents analogous run times for the test case B.

As we can see both xSVN repositories outperform normal SVN 1.6 for the commit operation for test cases A and B. That can be explained by the fact that the original version splits the documents in smaller chunks and writes them sequentially into Berkeley DB. So as many accesses to database for a single file are needed as many chunks represent the file. In turn, xSVN accumulates the whole content of an XML document in main memory and only then writes it into DB XML. The latter approach

<table>
<thead>
<tr>
<th></th>
<th>SVN 1.6</th>
<th>xSVN w/o indices</th>
<th>xSVN w/ indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit</td>
<td>1m32s</td>
<td>1m24s</td>
<td>1m27s</td>
</tr>
<tr>
<td>check out</td>
<td>1m05s</td>
<td>1m25s</td>
<td>2m18s</td>
</tr>
</tbody>
</table>

Table 8.1: Small Subset Test
### 8.3 Comparing to DB XML

<table>
<thead>
<tr>
<th></th>
<th>SVN 1.6</th>
<th>xSVN w/o indices</th>
<th>xSVN w/ indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit</td>
<td>100m17s</td>
<td>86m56s</td>
<td>93m17s</td>
</tr>
<tr>
<td>check out</td>
<td>3m24s</td>
<td>4m57s</td>
<td>5m1s</td>
</tr>
</tbody>
</table>

**Table 8.2: Big Subset Test**

However, has its natural drawback described in Section 8.2.1.

However, checking out run times are slightly lower for xSVN than in the SVN case. This can be explained by the following fact. SVN normally reads files by chunks retrieving chunks of certain fixed size from the Berkeley DB backend. This operation is highly optimized in Berkeley DB. In case of xSVN, the document content is read from the metadata field and then a necessary chunk is retrieved from the whole document. This may seem highly inefficient, but in practice according to our test cases it works reasonably well and is transparent in the scope of SVN Berkeley DB module in terms of implementation. Additionally, a big relative difference for the test case A can be explained by the fact that the opening of DB XML containers is a relatively expensive operation which takes most of the time when the retrieved content is small in size.

#### 8.2.1 Memory Usage in xSVN

One of the big difference between putting files into Berkeley DB and DB XML in xSVN is that in the former case it is done in a streaming fashion, whereas in the latter case the content of each document is accumulated in the main memory and only then is persisted in DB XML. This entails that xSVN needs as much memory as the size of the largest file to be committed. This is an architectural shortcoming of the xSVN architecture. However, it stems from a limitation in the DB XML API: DB XML is not able to add metadata fields in a streaming fashion. In xSVN’s DB XML container a dedicated metadata field is used for storing the original unparsed XML content. This speeds up retrieval times and allows to preserve XML formatting.

The shortcoming of high memory usage in case of large files can be eliminated by modifying DB XML in such a way so that it allows metadata streaming. From the author’s experience, XML documents rarely exceed 100MB, whereas TNTBase can easily cope with larger file depending on the amount of memory available in the system.

#### 8.3 Comparing to DB XML

The second important test scenario is to compare TNTBase to “pure” DB XML. Our goal was that integrating DB XML into SVN server in theory should not affect querying performance. To prove that we took TNTBase repositories for the test case B, with and without indices. Also we have manually created a DB XML container that contains the same XML documents. We have conducted performance measurements
on the following queries:

1. Getting names of the first 1000 documents:

   \[
   \text{fn:collection}()[\text{position()} < 1000]/\text{dbxml:metadata(}'\text{dbxml:name}'\text{')}
   \]

2. Retrieving a document by its name:

   \[
   (\text{: pure DB XML :})
   \text{fn:collection}()[\text{dbxml:metadata(}'\text{dbxml:name}'\text{')=} '\text{sets−operations.omdoc}'\text{']}
   \]

   \[
   (\text{: TNTBase :})
   \text{declare namespace} \ tnt='http://tntbase.mathweb.org/ns';
   \text{fn:collection}()[\text{dbxml:metadata(}'\text{tnt:file}'\text{')=} '\text{sets−operations.omdoc}'\text{']}
   \]

3. Searching for the union definition in the first 1400 documents:

   \[
   \text{declare default element namespace} 'http://omdoc.org/ns';
   \text{for} \ $a \ \text{in} \ \text{fn:collection}()[\text{position()} < 1400]
   //\text{theory[@xml:id='sets−operations']}//\text{definition}
   \text{return} \ $a[\text{tokenize(@for, } ', ')] = 'union'
   \]

We have selected the queries above since they mainly test the document retrieval from a database which is a major difference between retrieval scenarios in pure DB XML and TNTBASE. Upon retrieval an XQuery processor builds in memory an XDM instance representing obtained documents, so other operations on that XDM instance must be equally efficient in DB XML and TNTBASE. So we expect that if document retrieval by XQuery means is roughly the same in both systems then more complicated queries will also be similarly performant.

As we can see from Table 8.3, run times are approximately the same for pure DB XML and an xSVN repository without OMDoc-specific indices. In case of query 3, definition lookup runs much faster in the repository with indices which is an expected behavior since we had the indicies for the xml:id attributes and theory elements.
### 8.4 Other Tests

#### 8.4.1 Concurrent Access

In order to prove the TNTBase stability in the concurrent access scenarios, we have performed 2 tests: one is read-only, getting information via XQuery from multiple threads and getting data via checking out a working copy using an SVN interface of TNTBase. The second test dealt with committing files while querying a repository. In order to send querying requests we utilized Apache JMeter [Jme] – a Java-based desktop application for load testing functional behavior and measuring performance. As a repository to test we have chosen a TNTBase repository without OMDoc-specific indices with the full set of lecture notes.

**Read-only Test**  In this test we have constantly run the definition lookup query (query 3 from Section 8.3) and at the same time checked out the whole repository. Test scenario has been completed successfully. The results are the following:

- Query execution time slowed down twice, but stayed on the same level.
- The check out operation has completed in 7m16sec which is less than 2 times slower as in the case of pure check out (see Section 8.2).

As we can see, the slowdown is within an expected range since the I/O operations increased in approximately 2 times.

**Read/Write Test**  In this scenario we have run a definition lookup query constantly. For the test we have deliberately chosen the repository without OMDoc indices so that querying requires the whole container scan. At the same times we were trying to commit a single file and several XML documents multiple times. Here are the results on average:

- Regardless the amount of committed files, the commit time increased 3 times. The reason is that read/write locks that have to be acquired on the parts of the collection, and are mutually exclusive.
- Querying time during commit slows down proportionally to the amount of committed files. Internal deadlocks naturally occur in Berkeley DB and TNTBase implements the following resolution policy: transactions with less write operations are aborted and retried certain amount of times. If the limit of retries exceeds then the transaction is aborted. Practically since definition lookup without indices takes about 4s which involves the whole container scan, it is very probable that such a query will not succeed until a commit is done.

To resolve this a snapshot isolation transactions can be utilized: read transactions operate with a snapshot of data in such a way avoiding conflicts during write operations to the database. On the other hand, snapshot isolation requires more
resources, and since in our practice such conflicting situations occur rarely, it has been decided not to use snapshot isolation transactions in TNTBase, although it would be straightforward to add such a feature.

Additionally to the one commit/multiple queries scenario, we have made a test that constantly performs a definition lookup and tries to commit different files in parallel. During committing some lock conflicts occur inside Berkeley DB, but they are resolved transparently for a user and both commits succeed. If both commits take not so much time, then querying of the database is not interrupted, although is naturally slowed down.

We do not further expand on this issue since concurrent access to the database is a well studied topic in the database field and is out of the scope of this thesis.

8.4.2 Integrity Tests

Due to abnormal termination of some processes within TNTBase the following unfavorable situations may occur in theory:

1. Repository corruption when xSVN commit was aborted due to unpredictable circumstances, i.e. a network failure.

2. Inconsistencies in TNTBase FS (Section 3.2.3) due to incomplete commit.

3. Inconsistencies in query results while committing.

4. DB XML environment corruption and a necessity to run database recovery when a TNTBase process failed for some reasons.

Several tests have been performed in order to show that above situations are not possible:

1. During commit we tried to terminate the process on the client side as well as on the server side. Then we executed the svn verify utility on the repository. Corruption has never been the case.

2. Inconsistencies in TNTBase FS can never occur due to the following reason: an FS structure is updated in a single Berkeley DB transaction that is responsible for a final commit. If it is aborted or failed, an FS structure is not updated and a new revision is not persisted. If a new revision is committed without errors, then it entails that an FS structure has been successfully updated in the same transaction. This mechanism guarantees integrity of the file system in TNTBase.

3. Another potential problem that we have thought of is the case when a HEAD revision is being queried and at the same time a commit is occurring. A problem of consistency of query results arise. The consistency is guaranteed nonetheless due to the following reasons. A HEAD revision in TNTBase is identified by a −1 revision metadata field set on each document inside the DB XML container.
When new files are being committed they receive such a metadata field in the final Berkeley DB commit transaction, and such a metadata field is removed from old versions in the same transaction. Thus before the final Berkeley DB commit transaction old documents are accessed from a query since a new revision has not been yet updated for any of the documents. When revision metadata fields are updated, other query operations are blocked until the update is finished. If a Berkeley DB deadlock occurs, the deadlock resolution policy comes into play (refer to the Read/Write test on Page 167).

4. Sometimes corruption of Berkeley DB (and therefore DB XML) environment may occur due to some process failures. TNTBase takes care of them by implementing multi-process recovery mechanisms. Recovery of Berkeley DB environment from xSVN is inherited from SVN. In xAccessor an own mechanism is realized: when a xAccessor notices an environment failure it closes all opened DB XML handlers, recovers the environment by Berkeley DB means and retries all operations that are pending. Explicit tests were undertaken as well as natural tests during multiple years of TNTBase usage. TNTBase proved that it is able to successfully recover from Berkeley DB environment failures.

8.5 Problems in TNTBase

TNTBase has a number of known issues that stem from bugs or limitations in DB XML. This section describes them.

Metadata Streaming As was described in Section 8.2.1 DB XML does not support streaming of metadata which causes high memory usage upon committing when some files have considerable size. The way to fix high memory usage is to implement metadata streaming in DB XML, which is one of the new features of the upcoming DB XML release.

External Functions TNTBase makes extensive use of DB XML external functions. In TNTBase all external functions are implemented in Java. Java interface for implementing external functions have a number of known problems:

(1) Multi-threaded execution of the same external function causes an abnormal process termination.

(2) When returning a constructed document node from an external function, an empty sequence is returned.

(3) When returning a query result from external function which is obtained by executing another XQuery statement that returns a constructed XML element, a process exists abnormally.
8. PERFORMANCE AND STABILITY

(4) Usage of nested transactions in external function implementations causes an exception.

Problem 1 entails that external functions in TNTBase should be used with caution since simultaneous execution of the same external function may destroy a TNTBase instance. All other problems have been solved by the author in the cooperation with the DB XML developers.

XQuery Library Modules There are several problems with XQuery modules imports in DB XML that have been discovered by the author:

1. When trying to import a module that in turn imports another module with the same namespace DB XML reports circular imports which is not valid.

2. When trying to import a module that in turn imports another module with the same namespace and the same prefix DB XML reports “Prefix already bound error”. However, according to the XQuery W3C specifications the same prefixes are allowed in the module declaration and in the module import if they share the same namespace.

The above issues pose serious limitations on library modules usage in DB XML, and therefore in TNTBase. In particular, “private” and “public” TNTBase XQuery functions (such as tnt:doc(...) or tnt:commit(...)) had to be put into the same module to avoid the problems above. That entails that a user of TNTBase can potentially execute XQuery function that are not meant to be executed by an end-user.

“Complex” XQuery Queries DB XML proved to be quite unreliable when dealing with complex XQuery queries. For example, a tag library integration (See Chapter 5) failed due to segmentation faults caused by DB XML. Due to lack of time, we could not investigate those problems inside DB XML like we did before for other issues.

Infinitely Recursive Functions DB XML exist abnormally when executing infinitely recursive XQuery functions. For example,

```
declare function local:foo($i as xs:integer) {
    1 + local:foo($i)
};
local:foo()
```

Many XQuery processors handle this situation appropriately by throwing a corresponding exception or the like. Unfortunately, DB XML does not do that. This entails that execution of such queries in TNTBase will destroy a TNTBase process.

170
8.6 Conclusion

This chapter provided basic details about TNTBase performance, TNTBase technical characteristics as a software system as well as known issues and limitations.

We do not claim that we extensively covered problems of stability, performance and data integrity in TNTBase, although superficially covered the main topics and possible issues. This section rather shows that we are aware of the possible problems in the real world scenarios and know how to solve approach them. This section can also be considered as a pointer to what problems should be expected in your own VXD implementation. VDocs and tag libraries performance evaluation is out of the scope of this thesis.

Overall TNTBase is a mature software system that can be used in production reliably given some constraints on usage of certain features. Some functionality is still in the test mode and serves as a proof of concept, whereas functionality of VCS and native XML database is mature enough: we never experienced any loss of data over the years of TNTBase utilization.

In conclusion, with all its limitations TNTBase is designed without obvious technical drawbacks: there are just some technical issues in the underlying systems that can be fixed in practice.
8. PERFORMANCE AND STABILITY
Part V

Conclusion and Future Work
Chapter 9

Conclusion and Future Work

Not everything that is faced can be changed, but nothing can be changed until it is faced.

James Baldwin

9.1 Summary and Contributions

In this thesis we have covered multiple topics in the XML domain. We have looked in detail into XML versioning, XML querying and processing, XML database views and XML authoring and, as we have seen in Chapter 7, the concepts behind those topics can be highly applicable in the XML domain. Even though we had plenty of examples from the OMDoc field, the described concepts and their implementation – the TNTBase system – are suitable for other XML formats and XML in general.

A worked out rigorous theory for versioning allowed us to come up with a concept of a versioned XML database (VXD) - our foundation for solving further XML challenges. We have also presented engineering techniques that allowed to build a realization of the VXD concept that can serve as a basis for XML management systems and can offer multiple features for creating web-applications that deal with XML. We have learned that such a versioned XML database can be obtained by utilizing an existing stack of technologies by “combining” version control systems and XML-native databases. We have seen that XML versioning is not an easy problem and requires significant amount of formalizations to approach it “right”. One of the crucial optimizations both in concepts and in the engineering field would be to optimize querying access to previous revisions and possibly think of XML versioning on the level of objects, not files. Thorough reevaluation of the approach may be required. We have also learned that if an XML data store is designed in a proper way, it can significantly simplify adaptation to the XML language-specific domain and offload many format-specific tasks to the storage level (Chapter 6).
9. CONCLUSION AND FUTURE WORK

We have seen that the XML field can benefit from lessons learned in other fields. The work presented in this thesis contributes XML database views (virtual documents) learning from relational database views, tag libraries for XSLT and XQuery learning from Java tag libraries. Considerable modifications of the approaches have been required, but they allowed to adapt the ideas to the XML ecosystem without losing “an XML way” of dealing with problems.

As we promised in the introduction we partially achieved the high level goals from Section 1.3:

(G1) We enhanced XML storing by developing the VXD concept that focuses on XML history preservation without sacrificing XML access patterns.

(G2) We achieved more fine-grained separation between data and presentation by introducing the concept of XML database views where computational logic is clearly separated from the shape of the result document.

(G3) We improved authoring of XML documents by presenting the tag libraries approach where people with different skills can contribute to different parts of document design.

(G4) We facilitated data interchange between XML-driven applications by generalizing adaptation to a particular XML format and introducing the concept of F-methods where applications can exchange data by generating different forms of XML data presentation.

The concepts described in this work have been proven to be practical by the TNTBase system: They enable and simplify workflows that have been impossible or difficult before. TNTBase is much more than just a proof of concept or a mockup system. It has been reliably working for the last 3 years. It can be used as a standalone application for storing XML and non-XML content at the same time as well as be a part of other systems. One of the best examples is that TNTBase enabled the Planetary system by taking away one significant level of complexity – the storage level. TNTBase has exempted Planetary developers from thinking about storage and XML processing workflows:

- TNTBase supplies Planetary with the latest XML (OMDoc) content that can be pushed into a storage in two ways: via the Planetary web-interface or via an SVN client.

- TNTBase takes the responsibility to supply Planetary with previous versions of documents or SVN logs upon request, to provide features to operate with an underlying repository without a necessity to have a working copy.

- TNTBase takes care of XHTML generation from OMDoc sources, supplying Planetary with a prerequisites graph and building guided tours by means of virtual documents.
9.2 Future Work

- **TNTBase** offers querying possibilities that enable definition lookup, generation of the GenCS course book, concept dependency graphs, etc.

Of course, some configuration of **TNTBase** is needed for setting up validation and presentational workflows as well as providing some XQuery queries for performing fine-grained information extraction. But overall **TNTBase** acts as a “black box” for its users i.e. requires minimum amount of user intervention. It is available under the Apache 2.0 open-source license at [Tnt](#) together with documentation, links to the source code and the instructions how to build it.

### 9.2 Future Work

Here is the list of the future work items roughly ordered by importance:

**A Distributed Versioning Model** The next large step in the development of **TNTBase** would be an introduction of distribution facilities for versioned XML document storage supporting both push and pull-based workflows. We hope to gain not only distributed document management functionalities for **TNTBase**, but also to offer offline capabilities for web applications, which then can simply integrate the **TNTBase** library for transparent caching. In such applications, the **TNTBase** content would take the function of an SVN working copy with the additional ability of offline commits.

**Full XQuery Update Support** Modification of XML documents via XQuery Update in the scope of a VXD is still an open problem that has not been solved in this work, but some ideas are presented in Appendix B.4.

**Fine-grained Authorization** Fine-grained authorization and authentication on XML documents when using XQuery is still a problem to be studied. We made some work on this in [Gra+11](#) utilizing VDocs for post-filtering query results that a current requester can see. The mentioned paper described some approaches on modification of XML documents via XQuery Update, but that is still to be integrated into the VXD concept and implemented in **TNTBase** once the challenges mentioned have been approached.

**Tag Libraries Integration** The tag library approach described in Chapter 5 is rapidly maturing and tests are passed on quite elaborated examples. Although it was successfully tested on the Saxon XQuery processor and XQuery and XSLT reference implementations has been ported to the MarkLogic server by Erik Hennum, the XQuery implementation is still to be ported to **TNTBase**. Ideally, XQuery implementation should be platform-independent, i.e. it should run on every XQuery engine that supports XQuery 1.0. However, in practice almost every XQuery processor has its own problems. Unfortunately, DB XML is not an exception here. It exits abnormally even on simple tag documents. Once the bugs that cause it are fixed one could directly use XQuery implementation of the tag libraries approach in **TNTBase**.
9. CONCLUSION AND FUTURE WORK

Versioning of XML Elements  Even though VDocs provide fine-grained views on XML documents (by operating on XML objects) and enable editing facilities abstracting away from a notion of a file, the minimum versioned entity in TNTBase is still a file. However, it may be beneficial to have more fine-granular versioning. For instance:

- Get the earliest revision of a file where an attribute value became equal to some value.
- Get all revisions of a particular XML element.
- Get XML object differences between particular revisions.
- When a particular kind of change happened to a particular node.
- Ask for a revision in which a particular parameter has a particular value.

In order to achieve such finer granularity versioning on the element level, the key challenge is to understand how to preserve a node identity. For this a special versioned-aware client is needed like SVN does for usual files. Node identity can be also preserved, for instance, in special attributes, XML comments or processing instructions.

It may also be desirable to set up a minimum level of versioned objects. For example, in case of OMDoc we may want to treat theories as minimum versioned object, and then e.g. a change in the CMP element within a theory will result in showing that a corresponding theory element has been changed. We can define those levels based on depths or XPath expressions. This question is to be studied yet. There are many more open questions. For instance, how to determine a moved node or how to efficiently implement necessary algorithms for advanced differencing that take into account the node identity information preserved in XML documents.

Indexing Previous Revisions  At the moment accessing previous revisions in TNTBase does not benefit from indices presented in the database. That may be a serious performance bottleneck if we try to access history from XQuery relatively often. Unfortunately, this is not easily possible (if at all) without modifying the underlying XML DB (in our case, DB XML). This improvement is to be studied yet and the ways how to achieve this should be explored. On a very abstract level it is clear that a modified DB XML version should be aware of the xSVN backend storage module. This information should be utilized to store indices for previous revisions although they are represented not as fulltext but deltas against newer revisions.

An xSVN Upgrade  xSVN is based on SVN 1.4. While xSVN in terms of performance is not worse than SVN 1.6 (see Section [8.2]), some features from SVN 1.6 may be useful: e.g better merging handling or additional consistency checks. Therefore we have plans to port xSVN to the SVN 1.6 code base. Since only the Berkeley DB module of SVN has been modified to implement xSVN, it should be relatively straightforward to accomplish this task.
9.2 Future Work

Despite many open problems and possible improvements to be done, at the moment of writing TNTBase is still up and running serving as an XML backend storage for some of KWARC group projects without almost any maintenance costs.
9. CONCLUSION AND FUTURE WORK
Part VI

Appendix
APPENDIX A

NOTATIONS OVERVIEW AND ABBREVIATIONS

A.1 Notations Overview

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ℬ</td>
<td>The set of bytes (cf. page 14)</td>
</tr>
<tr>
<td>ℐ</td>
<td>The set of identifiers (cf. page 14)</td>
</tr>
<tr>
<td>™</td>
<td>The set of data store keys (cf. page 14)</td>
</tr>
<tr>
<td>™</td>
<td>The set of metadata keys (cf. page 14)</td>
</tr>
<tr>
<td>ℳ</td>
<td>The set of names (cf. page 14)</td>
</tr>
<tr>
<td>ℓ</td>
<td>A data store (cf. page 14)</td>
</tr>
<tr>
<td>ℰ</td>
<td>A data store function (cf. page 14)</td>
</tr>
<tr>
<td>ℰ</td>
<td>The set of all data stores (cf. page 14)</td>
</tr>
<tr>
<td>🏢</td>
<td>An FS-tree (cf. page 15)</td>
</tr>
<tr>
<td>ℳ</td>
<td>A set of directory nodes of an FS-tree (cf. page 15)</td>
</tr>
<tr>
<td>℺</td>
<td>A set of file nodes of an FS-tree (cf. page 15)</td>
</tr>
<tr>
<td>ℱ</td>
<td>A set of edges of an FS-tree (cf. page 15)</td>
</tr>
<tr>
<td>ℱ</td>
<td>A root node of an FS-tree (cf. page 15)</td>
</tr>
<tr>
<td>ℱ</td>
<td>A name labeling function of an FS-tree (cf. page 15)</td>
</tr>
<tr>
<td>ℰ</td>
<td>A content function of an FS-tree (cf. page 15)</td>
</tr>
<tr>
<td>ℱ</td>
<td>The set of all FS-trees (cf. page 15)</td>
</tr>
<tr>
<td>ℱ</td>
<td>An FS-tree path function (cf. page 15)</td>
</tr>
<tr>
<td>ℱ</td>
<td>The set of all FS-tree paths (cf. page 15)</td>
</tr>
<tr>
<td>φ</td>
<td>An fs-mapping function of an FS-enabled data store (cf. page 15)</td>
</tr>
<tr>
<td>(δ, π)</td>
<td>A pair of diff/patch functions (cf. page 16)</td>
</tr>
<tr>
<td>ℭ</td>
<td>A revision tree (cf. page 17)</td>
</tr>
<tr>
<td>η</td>
<td>A node identification function (cf. page 17)</td>
</tr>
<tr>
<td>ρ</td>
<td>A revision number or a revision function of a revision tree (cf. page 17)</td>
</tr>
<tr>
<td>η</td>
<td>A node identification function (cf. page 17)</td>
</tr>
</tbody>
</table>
### A. NOTATIONS OVERVIEW AND ABBREVIATIONS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau$</td>
<td>A node type function (cf. page 17)</td>
</tr>
<tr>
<td>$\chi$</td>
<td>A copy identifier of a node (cf. page 17)</td>
</tr>
<tr>
<td>$\xi$</td>
<td>The set of node types (cf. page 17)</td>
</tr>
<tr>
<td>$\iota$</td>
<td>The set of node revision identifiers (cf. page 17)</td>
</tr>
<tr>
<td>$\mathcal{R}$</td>
<td>A repository (cf. page 18)</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>A set of repository cross-revision edges (cf. page 18)</td>
</tr>
<tr>
<td>$V_\mathcal{R}$</td>
<td>A set of repository vertices (cf. page 18)</td>
</tr>
<tr>
<td>$D_\mathcal{R}$</td>
<td>A set of repository directory nodes (cf. page 18)</td>
</tr>
<tr>
<td>$F_\mathcal{R}$</td>
<td>A set of repository file nodes (cf. page 18)</td>
</tr>
<tr>
<td>$E_\mathcal{R}$</td>
<td>A set of repository edges (cf. page 18)</td>
</tr>
<tr>
<td>$\eta_\mathcal{R}$</td>
<td>A node revision identification function (cf. page 18)</td>
</tr>
<tr>
<td>$\psi$</td>
<td>A copy identification function (cf. page 18)</td>
</tr>
<tr>
<td>$s$</td>
<td>A node successor function (cf. page 18)</td>
</tr>
<tr>
<td>$\mathcal{R}_\text{HEAD}$</td>
<td>A HEAD revision tree (cf. page 18)</td>
</tr>
<tr>
<td>$s$</td>
<td>A node successor function (cf. page 18)</td>
</tr>
<tr>
<td>$\mathcal{R}$</td>
<td>The set of all repositories (cf. page 18)</td>
</tr>
<tr>
<td>$\ell$</td>
<td>A node revision lookup function (cf. page 19)</td>
</tr>
<tr>
<td>$c^f$</td>
<td>A fulltext content function (cf. page 19)</td>
</tr>
<tr>
<td>$\zeta$</td>
<td>A checkout node labeling function (cf. page 20)</td>
</tr>
<tr>
<td>$\tau^c_\mathcal{R}$</td>
<td>A checked out FS-tree (cf. page 20)</td>
</tr>
<tr>
<td>$\mathcal{C}$</td>
<td>A repository creation function (cf. page 32)</td>
</tr>
<tr>
<td>$D$</td>
<td>A data store repository (cf. page 34)</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>A set of data stores of a data store repository (cf. page 34)</td>
</tr>
<tr>
<td>$d$</td>
<td>A data store function (cf. page 34)</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>A key-mapping function (cf. page 34)</td>
</tr>
<tr>
<td>$X$</td>
<td>The set of XDM instances (cf. page 35)</td>
</tr>
<tr>
<td>$X_D$</td>
<td>The set of XDM document nodes (cf. page 35)</td>
</tr>
<tr>
<td>$\mathcal{X}$</td>
<td>An XML data store (cf. page 35)</td>
</tr>
<tr>
<td>$\mathcal{X}$</td>
<td>The set of all XML data stores (cf. page 35)</td>
</tr>
<tr>
<td>$\mathcal{X}_T$</td>
<td>An XML FS-subtree (cf. page 35)</td>
</tr>
<tr>
<td>$\mathcal{X}$</td>
<td>An XML-enabled repository (cf. page 35)</td>
</tr>
<tr>
<td>$XQ$</td>
<td>The set of strings of the XQuery language (cf. page 47)</td>
</tr>
<tr>
<td>$\xi$</td>
<td>An XQuery processor function (cf. page 47)</td>
</tr>
<tr>
<td>$\mathcal{XD}$</td>
<td>An XML database (cf. page 48)</td>
</tr>
<tr>
<td>$\mathcal{N}$</td>
<td>An XML database default namespace (cf. page 48)</td>
</tr>
<tr>
<td>$\mathcal{U}$</td>
<td>The set of URIs (cf. page 48)</td>
</tr>
<tr>
<td>$\omega$</td>
<td>A key-mapping function (cf. page 48)</td>
</tr>
<tr>
<td>$M$</td>
<td>The set of metadata keys (cf. page 50)</td>
</tr>
<tr>
<td>$\mu$</td>
<td>A data store metadata function (cf. page 50)</td>
</tr>
<tr>
<td>$\theta$</td>
<td>A path mapping function (cf. page 51)</td>
</tr>
</tbody>
</table>
### A.2 Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Expansion</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCS</td>
<td>A version control system</td>
<td></td>
</tr>
<tr>
<td>XeR</td>
<td>An XML-enabled repository</td>
<td>cf. Chapter 2</td>
</tr>
<tr>
<td>XML DB</td>
<td>An XML-database</td>
<td></td>
</tr>
<tr>
<td>VXD</td>
<td>A versioned XML-database</td>
<td>cf. Chapter 3</td>
</tr>
<tr>
<td>FS</td>
<td>A file system</td>
<td></td>
</tr>
<tr>
<td>FSD</td>
<td>A file system document</td>
<td>An XML document that represents directory contents</td>
</tr>
<tr>
<td>BDB</td>
<td>Berkeley DB</td>
<td></td>
</tr>
<tr>
<td>DB XML</td>
<td>Berkeley DB XML</td>
<td></td>
</tr>
<tr>
<td>VDoc</td>
<td>A virtual document</td>
<td>The concept of XML database views</td>
</tr>
<tr>
<td>VDFSE</td>
<td>A VDoc file system entity</td>
<td></td>
</tr>
<tr>
<td>VDP</td>
<td>A virtual document processor</td>
<td>An implementation of the VDoc concept</td>
</tr>
<tr>
<td>VDoc Spec</td>
<td>A virtual document specification</td>
<td>An XML view definition</td>
</tr>
<tr>
<td>VDoc Skel</td>
<td>A virtual document skeleton</td>
<td>A part of a VDoc Spec</td>
</tr>
<tr>
<td>CA</td>
<td>A configuration area</td>
<td>A location in a VCS that defines F-methods</td>
</tr>
<tr>
<td>PP</td>
<td>A processing property</td>
<td>A VCS property that defines associations between F-methods and files or folders</td>
</tr>
</tbody>
</table>
A. NOTATIONS OVERVIEW AND ABBREVIATIONS
Appendix B

Building Bridges between Theory and Practice

B.1 What Should be Considered as XML for an XeR

We envision several basic approaches of defining a data store function \( d \) (recall Definition 2.22) in practice. Section 2.5.1 provides an example of how it is done in xSVN. Essentially, defining that function directly influences what files within a repository will be exposed to the XML functionality of a VXD.

1. A set of predefined file extensions/patterns that are treated as an indication that a file in question is in the XML format. This set can be hard-coded into an XeR or/and specified in some form of the configuration area (e.g. files) understandable by an XeR. For example, in TNTBase files with extensions omdoc and xml are automatically considered as XML files.

2. Some metadata associated with a file can prompt an XeR that a particular file is supposed to be XML. An obvious metadata field would be a media type \([\text{FB96}]\). A media type can be defined via \( V \)'s working copy properties if there is a support for them. For instance, SVN has a built-in reserved property name \( \text{svn:mime-type} \)\(^1\) which can be used for determining whether a particular file is in the XML format. However, any user-defined property can be used for such a purpose. A set of such properties can be predefined in an XeR or be defined/overridden in the XeR configuration area analogous to the case with file extensions/patterns.

3. Use some heuristic to define whether a committed file is in the XML format, e.g. presence of an XML processing instruction \(<\!\?xml\ version='"1.0"' ?\>\).

4. Try to parse every file and check whether it is well-formed XML. If it is, we should treat it as an XML file.

\(^{1}\text{Media type was called MIME type formerly}\)
B. BUILDING BRIDGES BETWEEN THEORY AND PRACTICE

First three approaches do not guarantee that a file mapped to an XML data store is indeed XML. In this case an underlying XML data store will raise an error condition and an XeR will abort a commit transaction. The fourth approach is more expensive efficiency-wise than the others.

B.2 What Can Simplify Integration of an XML DB into a VCS

There are a number of system characteristics that can significantly simplify integration of an XML DB into a VCS.

Let us start with VCSs:

1. A VCS must conform to a versioning model introduced in Chapter 2. Otherwise, more complex adaptations of the VCS are expected.

2. If a VCS has a clear architectural separation between storage module and other modules, then obviously it becomes easier to integrate an XML DB into it. This is a common practice for software engineering, and therefore we assume that any wide-spread VCS is designed appropriately.

3. Most VCSs support user-defined metadata on files and directories which are most commonly called properties. The XeR concept itself does not make use of VCS properties, but as exemplified in Section 6.2.1.4, properties can significantly help in plugin automatization in the scope of the VXD(F) concept.

From the XML DBs side the list is the following:

1. XML DB embeddedness can save some efforts on realizing a module for intercommunication between a VCS storage module and an XML DB itself. Embeddedness means that an XML DB can be integrated into another application process (e.g. when an XML DB is assembled as a dynamic linked library (DLL) in case of C++ or a JAR file in case of Java).

2. The VXD approach uses an ability to associate metadata with XML documents (Chapter 3). If an XML DB does not support metadata, then additional steps are required to “emulate” this functionality (refer to Appendix B.3).

3. Transaction support in an XML DB can diminish efforts spent on guaranteeing consistency and integrity of data. In some case it is even possible to share internal transactions between an XML DB and a VCS. DB XML and SVN are such examples.

4. Support for XQuery external functions can make XQL capabilities much richer by supplying XQuery functions that are not easily implementable (if at all) in pure XQuery. Examples of such functions are: accessing previous revisions of XML documents, retrieving non-XML content, committing documents.
B.3 Any XML Database is Metadata-enabled

If an XML DB does not support metadata, they can be “emulated” in it. We envision that we can embed metadata in separate documents that are linked to the original XML documents. Since each document in an XML DB has its URI, this URI can serve as a linking part to the document metadata.

Let us provide an example: Assume that a document has the URI ns://doc1 (i.e. it can be retrieved via the XQuery function fn:doc) and is stored in the XML DB. Then we can store any metadata for this document in another document of the form:

```xml
<vx:metadoc uri='ns://doc1'/>
<vx:metadata name="path">
lectures/graphs.OMdoc
</vx:metadata>
...
<vx:metadata ... />
</vx:metadoc>
```

Then the XQuery function ns:metadata from Definition 3.3 can be implemented as follows:

```xml
declare function ns:metadata($m as xs:string, $d as node()) as item()? {
  fn:collection()/vx:metadoc[@uri=fn:document-uri($d)]/vx:metadata[@name=$m]/node()
};
```

Thus we provided a metadata XQuery function and therefore according to Definition 3.3 the XML DB is metadata-enabled although it does not support metadata out of the box. A modified VCS V must be responsible for creating such documents when modifying the repository’s content. For instructions see Section 3.2.2.

B.4 XQuery Update with a Temporal Aspect

This question still has to be studied in more detail. The problem is that we have to preserve history when changing documents using XQuery Update (XQU). Since XQU modifies documents directly in  ucwords(X), we may not use it directly. The idea of approaching this problem is to obtain modified documents as a result of an XQU expression and then use VcsAccessor to commit changes back to the repository. To achieve this we envision several approaches:

- Transform any XQuery Update expression to an XQuery Update transform function. A transform function is a function that copies a node, modifies it in memory and returns it, unlike a pure XQU expression that never returns a value. A drawback of this approach is that it is not a trivial task to do such query transformations. Some work on this topic has been done in [Ref10] and [FCB07]. Additionally, transform functions return only one value, which means that only one document can be modified in a time. Therefore we should transform a XQU query to multiple transform functions and aggregate the results.
B. BUILDING BRIDGES BETWEEN THEORY AND PRACTICE

• Create special extension functions that will take a document node to be modified and additional parameters for the corresponding XQU method. These functions would commit from XQuery itself using an external commit function, or return results and some method of xAccessor will pick the results and proceed with committing via VcsAccessor. The question how to aggregate information from multiple functions arises. The powerful concept of PULs (pending update lists) is no longer applicable here since we cannot reach parallelization in pure XQuery.

The first approach seems to be “cleaner” and can be realized by leveraging the work referenced above.

B.5 REST Interface for a VXD

The way a developer designs an interface for his own VXD implementation is not bound to any particular technology. We have experimented and implemented a REST interface for TNTBase, and introduced the following REST resources:

• A Document Resource: Retrieving documents from different revisions including VDocs. Committing new/modified documents is also possible as well as deleting. All modification operations are done without a working copy so that other applications can modify content without a necessity to check out their own working copy.

• A File System Resource: Retrieving file systems information in different formats (e.g. XML or HTML). For example, figuring out what folders/documents/VDocs a particular directory contains.

• A Querying Resource: An ability to submit any XQuery expression and retrieve results. All XQuery functions from an XQL should be accessible. For example, count elements in a particular VDoc and commit those counters as a text file.\footnote{If a VXD has an XQuery external commit function (like TNTBase)}

• A Caching Resource: Caching previous revisions for fast access and retrieval as described in Section 3.3.2.2. This resource is also responsible for cleaning cache that is no longer in use.

• A VDocs Resource If a VXD implements a virtual document concept (Chapter 4), this resource should be responsible for creating/updating/deleting VDoc FS entities. Additional REST methods may be provided, for instance, for retrieving VDoc content based on a VDoc Spec only. This resource should take care of VDocs materialization (Section 4.3.3).

• An $\mathcal{F}$-Specific Resource If a VXD implements the VXD($\mathcal{F}$) concept then this resource is responsible for executing $\mathcal{F}$-methods and caching/retrieving $\mathcal{F}$-results.
• **A Helper Resource** It can be anything that helps administering a VXD or gets some kind of information about repository. For example, it could be REST methods for managing indexing information, getting statistics about number of documents or last modification timestamps. FS consistency checks may be a part of this resource as well.

### B.6 XQuery External Functions

Many XML databases provide the possibility to implement custom XQuery functions in other programming languages like Java or C++ and bind them to an XQuery function signature.

Having such a possibility allows us to implement additional XQuery functions that make use of the VcsAccessor interface for e.g. retrieving revision logs. Implementation of such functions is an essential part of xAccessors. However, use cases of VXD external functions are not limited to VCS functionality only. Here is the list of possible classes of external functions use cases in VXD:

- Retrieving previous revisions of XML documents or non-XML content of text-based format for various revisions.
- Retrieving underlying repository information. E.g. a change log or a list of revisions in which a particular set of documents has been modified.
- Modifying repository content via XQuery expressions, i.e. allowing documents to be committed directly from XQuery code.
- XQuery expressions dynamic evaluation useful in e.g. implementation of a VDP (Section 4.3).
- Execution of \( \mathcal{F} \)-methods directly from XQuery expressions. That allows a user to operate with \( \mathcal{F} \)-results without caching them first.

It is very important to realize that all external XQuery functions may have side-effects. In particular, this means that it is not guaranteed that an external function will behave in the same way when invoked multiple times on the same input data in the same query (unlike pure XQuery functions). Developers should take this into account.

### B.7 Why not XML-diff in the First Place

XML-diff algorithms are intentionally not in the core of the VXD concept. Some reasons have been given in Section 2.5.2 on the TNTBase example, but they also hold for any VXD in general.

Text-based differencing algorithms are very efficient, and therefore XML-diff brings more advantages to users than to the efficiency of VXD implementations. Furthermore, text-based diff is typically more user-readable that XML-diff (given that e.g. indentation of files has not been changed).
B. BUILDING BRIDGES BETWEEN THEORY AND PRACTICE

For example, if a new one-line element has been added to an XML file. The XML-diff fragment in the form of XQuery Update can look like:

\[
\text{insert nodes } <foo/> \text{ as last into } /doc[1]/bar[14]/baz[82]
\]

The snippet above in many cases is not that intuitive in comparison to the text-diff when an added fragment and its file position is shown. Of course, text-diff does not expose structural information, which is one of the reasons to have XML-diff. However, pure XML-diff is often too verbose and unspecific since when dealing with a concrete format users are typically not interested in difference details on the XML node level. Instead it is beneficial to see differences on a format level. For example, in OMDoc a user might be interested which theorems, examples or sections have been changed.

Therefore the following conclusions have been made:

- Deeply format-specific XML-differencing algorithms are typically more useful.
- XML-diff algorithms neither affect nor contribute to the VXD concept.
- XML-diff capabilities can be highly beneficial for an end-user and must not be tied with the core of the VXD implementation.
- An ability to integrate third-party XML diff implementations into a VXD is extremely important. A particularly interesting example of such implementation that utilizes format-specific models for comparing XML can be found in [Mü08].
Appendix C

The TNTBase XQuery Library

The list below presents the most important XQuery functions that constitute the TNT-Base XQL. Due to DB XML problems with multiple module namespace declaration, we have the same prefix for pure and external XQuery functions. However, it is clear from the signature which functions are external.

- \texttt{tnt:doc(path as xs:string?)}
  
  Retrieves a physical XML document or a virtual document based on its path.

- \texttt{tnt:doc(path as xs:string?, rev as xs:integer) as document−node()?}
  
  Retrieves an XML document based on its path and a revision number.

- \texttt{tnt:txn−doc(path as xs:string?, $txn as xs:string) as document−node()?}
  
  Retrieves a document based on its path and an SVN transaction identifier.

- \texttt{tnt:collection() as document−node()∗}
  
  Retrieves all XML documents from the HEAD revision.

- \texttt{tnt:collection(file−pattern as xs:string) as document−node()∗}
  
  Retrieves XML documents from HEAD revision based on a file pattern.

- \texttt{tnt:collection(rev as xs:integer) as document−node()∗}
  
  Retrieves a cached collection of XML documents for a particular revision.

- \texttt{tnt:get−path(node as node()) as xs:string?}
  
  Returns a path of a document in the database to which the node belongs.

- \texttt{tnt:get−revision(node as node()) as xs:integer?}
  
  Returns a revision number of the node (−1 for HEAD, and a positive number if a node belongs to a document that has been cached for accessing previous revisions).

- \texttt{tnt:commit(paths as xs:string∗, docs as element()∗, msg as xs:string) as xs:string external}
  
  Commits a multiple documents in the same commit transaction.
C. THE TNTBASE XQUERY LIBRARY

- **tnt:random($range as xs:integer) as xs:double external**
  Returns a random floating-point number

- **tnt:execute($query as xs:string, $prefixes as xs:string*, $nss as xs:string*) as item()* external**
  Executes an XQuery expression represented as a string. Bound prefixes and namespaces are the optional arguments.

- **tnt:file-revs($path as xs:string) as xs:integer* external**
  Returns revision numbers where a document under a certain path has been modified.

- **declare function tnt:last-revnum() as xs:integer**
  Returns the revision number of the HEAD revision

- **tnt:tnt-diff($e1 as element(), $e2 as element())**
  Returns an XML-diff between two XML elements.

- **tnt:file($path as xs:string?, $revision as xs:integer?) as xs:string? external**
  Returns a text content of any file (also non-XML) from a repository.

- **tnt:f-result($path as xs:string, $name as xs:string+) as document-node()?**
  Returns an F-results (if cached) based on a path and an F-method name.

- **tnt:vdoc-spec-expand($path as xs:string) as node()**
  Retrieves VDoc content based on the VDoc Spec only (referenced by path)

- **tnt:submit-vdoc($path as xs:string, $vdoc as element(), $msg as xs:string)**
  Submits an edited VDoc based on its path and its modified version. The third parameter receives a commit message that will be used upon commit if no immutable nodes were changed.

We listed only the most important functions. We can observe that the functions that require the SVNAccessor functionality are implemented as external functions.
Bibliography


[Jer] *Reference Implementation for building RESTful Web services*. URL: `https://jersey.dev.java.net/` (visited on 05/01/2012) (cit. on p. 64).


Semantic Markup for LATEX. Project Homepage. URL: http://trac.kwarc.info/sTeX/ (visited on 02/22/2011) (cit. on p. 150).

SVNKit – The only pure Java Subversion library in the world! URL: http://svnkit.com/ (visited on 05/01/2012) (cit. on pp. 60, 64).


BIBLIOGRAPHY


BIBLIOGRAPHY


