Scripting Documents with XQuery: Virtual Documents in TNTBase

Vyacheslav Zholudev, Michael Kohlhase
Jacobs University Bremen
{v.zholudev,m.kohlhase}@jacobs-university.de

April 16, 2010

Note for reviewers: *With permission of Ms. Usdin, we are providing the paper in PDF for reviewing. If our paper gets accepted, we commit ourselves to submit a final version in XML following all instructions given at the Balisage web-page.*

Abstract

This paper introduces the concept of Virtual Documents and its prototypical realization in our TNTBase system, a versioned XML database. VDs integrate XQuery-based computational facilities into documents like JSP/PHP do for relational queries. We view the integration of computation in documents as an *enabling technology* and evaluate it on a handfull of real-world use cases.

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[1] 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 paper, we present a *Virtual Documents* (VDs), a general framework for *integrating XQueries into XML documents as computational devices* and processing them efficiently. As a rough approximation, VDs 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,

[1] 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.

intro.tex 621 2010-04-16 13:51:04Z vzholudev

TNTBase Virtual Documents are the results of XQueries computed on demand from the XML files explicitly represented in TNTBase, presented to the user as entities (files) in the TNTBase file system. Like views in relational databases TNTBase VDs are editable, and the TNTBase system transparently patches the differences into the original files in its underlying versioning system. Thus 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. Again, like relational database views, VDs become very useful abstractions in the interaction with versioned XML storage.

We have already discussed VDs [ZKR10], concentrating on theoretical issues such as when XML-based document formats admit virtual documents. Since then, our VDs implementation has been extended and matured considerably, and we will concentrate on features and real word use cases and practical issues. In the next section we recap the basics of our TNTBase system, before we introduce the functionality of VDs in Section 3. Section 4 discusses some high-profile use cases of VDs and Section 5 concludes the paper.

2 TNTBase, a Short Recap

The TNTBase system is a versioned XML-database with a client-server architecture. Essentially, it consists of two parts: the core and the application-specific layer. Let us briefly discuss them.

2.1 The Core

The core of TNTBase consists of the xSVN module, which integrates Berkeley DB XML [Ber09b] into a Subversion server [SVN08]. DB XML stores HEAD revisions of XML files; non-XML content like PDF, images or LaTeX source files, differences between revisions, directory entry lists and other repository information are retained in a usual SVN back-end storage (Berkeley DB [Ber09a] in our case). It is worth mentioning here that TNTBase also supports ***branching*** as SVN does (see also Section 4.2 for a use-case of virtual documents with on this). Keeping XML documents in DB XML allows us to access those files not only via any SVN client, but also through the DB XML API that supports efficient querying of XML content via XQuery [BCF+07] and modifying that content via XQuery Update [CDF+08]. As many XML-native databases, DB XML (and hence TNTBase) supports ***indexing***, which improves performance of certain queries. TNTBase also adopted ***transactional support*** (atomicity, consistency, isolation, durability) from DB XML.

The TNTBase system is realized as a web-application that provides two different interfaces to communicate with: an xSVN interface and a RESTful interface (for details refer to [Z+10]) for XML-related tasks. The xSVN interface behaves like the normal SVN interface — the mod_dav_svn Apache module serves requests from remote SVN clients — with one exception: If one of the committed XML files is ill-formed, then xSVN will abort the whole transaction. The RESTful [JSR09] interface provides XML fragment access to the versioned collection of documents:

Querying: As every XML-native database, TNTBase supports XQuery, but extends the DB XML syntax by a notion of file system path and revision to address different versions of path-based collections of documents.

Modifying: Apart from modifying any kinds of documents via any SVN client, TNTBase takes advantage of XQuery Update facilities, and, in contrast to pure DB XML, modified documents are versioned, i.e., a new revision is committed to xSVN whenever some changed are made to the documents stored in a TNTBase repository .

Querying of previous revisions: Although xSVN's DB XML back-end by default holds only HEAD (the last) revisions of XML documents, and others are stored as reverse deltas against HEAD revisions, it is also possible to access and query previous versions by additionally providing a revision number to the TNTBase XQuery extension functions. It is necessary to note, that previous versions cannot be modified since once a revision is committed to an xSVN repository, it becomes persistent.

Virtual Files: This is a precursor technology to the Virtual Documents discussed in this paper. It has been described in detail in [ZK09]: a Virtual File is a TNTBase file system entity whose content is defined by a single XQuery expression. For the end user they are like normal files whose content are wrapped results of an associated query. Virtual Files also can be queries and modified.

For more information about the TNTBase core refer to [ZK09]. Since then we have significantly increased stability and performance that can be proved that TNTBase is being used for the LATIN Project [LAT], for General Computer Science lectures repository (that counts more than 2000 XML documents with over 2500 revisions) and Translation SUMO to OMDoc Project [Mis10]. Moreover, we are mirroring some of TNTBase repositories to normal SVN repositories by replication functionality adopted from Subversion. This possibility once more justifies the decision of combining the two systems. If something went wrong with a TNTBase repository and the data got corrupted (actually, it never happened to us), then we can easily restore them from a replicated SVN repository with all history preservation.

2.2 Application Layer of TNTBase

In our experiments it turned out that many tasks specific to particular XML formats can be done by TNTBase, and that was a reason to derive a separate layer on top of the TNTBase core and augment this layer with format-specific functionalities (see Figure 1). Although the detailed information can be found in [ZKR10], let us briefly describe the major features:

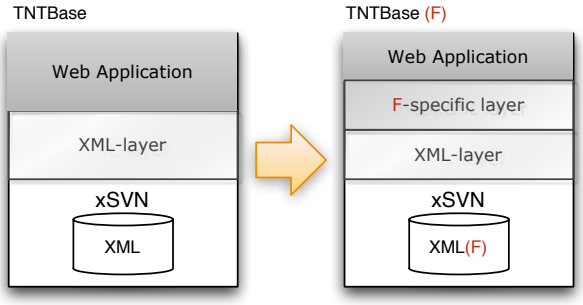

Figure 1: TNTBase

Validation, Presentation, Interface Extraction, Plugin Architecture TNTBase provides facilities to integrate format-specific validation (e.g. for RelaxNG schemas) and presentation (e.g. via XSL transformations). But sometimes a format requires more specific functionality, e.g. extraction and caching RDF information upon commit or retrieval of rendered MathML. Such functionality can be supplied as additional modules and injected into the application layer via the TNTBase plugin API. Configuration files are also stored in a TNTBase repository, so a user do not have to have an access to a server: for instance, commit-time behavior is defined by an SVN *tntbase:validate* property that can be assigned to

files as well as to whole directories. Pre-commit or post-commit hooks that are automatically generated take care of processing committed information based on the configuration files. In case of pre-commit processing a corresponding plugin has access to the documents that are about to be committed, and may reject a transaction if the collection of committed documents is format-inconsistent, or clashes with existing documents in the repository. Last but not least, TNTBase RESTful URLs that are used to perform validation or presentation are dynamically changed once configuration files are modified.

Custom XQuery modules A user can write his own XQuery extensions and store them in the repository. Thus it is not necessary to have modules located in the server's file system or remotely. XQuery modules can be referenced inside repository itself, which might happen to be useful if the development of XQuery modules is still in progress.

Virtual Documents Virtual Documents are also a part of an application layer, but we will focus on them in the next sections.

TNTBase also gained number of features that have been requested by TNTBase users. To name just a few of them: integration with JOBAD framework [JOB08], extracting RDF from OMDoc [Koh10] documents upon commit and storing it in TNTBase, integration with LaTeXML [Mil10] and Virtuoso [Ope] (for more details refer to [DKL+10]). Those features are pluggable, so a new TNTBase installation does not have to include them.

3 Virtual Documents

This section introduces practical aspects and technical details of Virtual Documents (for the theory we refer to [ZKR10]) using a simple running example to fortify our intuitions. Section 4 will tackle the real-world scenarios and justify VDs in TNTBase.

VDs are the first class citizens in the TNTBase file system. Whereas internally they are quite different from usual documents in the repository, for a user they look like normal files: one can browse them, validate, apply stylesheets, query and even modify. VDs are essentially a tight mix of static XML parts with XQuery queries and instructions in XML form that prompt TNTBase how to organize the XQuery results inside a VD. Let us start with a simple example. Assume that we want to have a joined list of mathematical exercises together with authors contributed to them. We might want to have the root element and the elements that embrace authors and exercises (we will refer to Figure 2 throughout this section). XQueries that select necessary data will augment our document. The way we describe the VD is the subject of the next subsection.

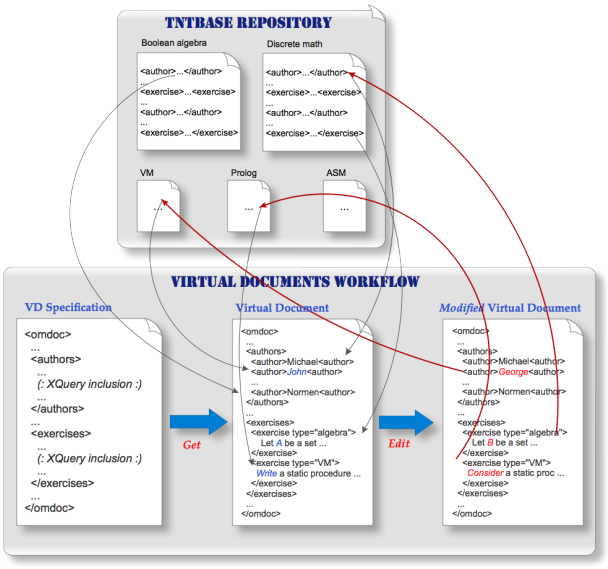

Figure 2: VD workflow

3.1 VD Specifications and Skeletons

VD Specification (VD Spec) is the most important part of any VD. Basically, it is a document template with XQuery inclusions and some other auxiliary elements helping TNTBase to figure out how to execute a particular XQuery and how to populate query results. It must be XML. In Listing 1 one can see a simple example of a VD Spec that is supposed to define a VD that would contain thematic lecture exercises together with their authors. For the complete RelaxNG [Rel] schema refer to [ZK].

Listing 1: Example of a VD Spec

```
<tnt:virtualdocument xmlns:tnt="http://tntbase.mathweb.org/ns">
  <tnt: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:
          <tnt:xqinclude query="tnt:collection('/exercises//*.omdoc')//dc:creator">
            <tnt:return><tnt:result/></tnt:return>
          </tnt:xqinclude>
        </omtext>
      </omdoc>
      <omdoc>
        <dc:title>Exercises</dc:title>
        <tnt:xqinclude>
          <tnt:query name="exercises.xq"/>
          <tnt:return><tnt:result/></tnt:return>
        </tnt:xqinclude>
      </omdoc>
    </omdoc>
  </tnt:skeleton>

  <tnt:query name="exercises.xq">
    for $t in $topics return
      tnt:collection(concat('/exercises/', $t, '/*.omdoc'))//exercise[position() le $max]
  </tnt:query>

  <tnt:params>
    <tnt:param name="max">
      <tnt:value>10</tnt:value>
    </tnt:param>
    <tnt:param name = "topics">
      <tnt:value>search</tnt:value>
      <tnt:value>graphs</tnt:value>
    </tnt:param>
  </tnt:params>
</tnt:virtualdocument>
```

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

VD Skeletons contain a mixture between any XML nodes and **tnt:xqinclude** elements. The latter ones specify a single XQuery query and "the rules" how results of that query will be mixed with other elements in a VD. The rules are enclosed into a single **tnt:return** child element that, in turn, contains a mixture of any XML elements with empty **tnt:result** elements. In order to understand how VD content is produced let us consider the following sequence of

actions:

1. We take a **tnt:xqinclude** element and obtain an XQuery associated with it
2. We get the results of that query and iterate over them
3. For every result we get children of the **tnt:return** element and substitute any **tnt:result** element with a considered query result
4. We concatenate all children obtained from step 3) in order
5. The result of concatenation replaces the considered **tnt:xqinclude** element
6. Repeat steps 1)-5) for all **tnt:xqinclude** elements in a VD Spec

Although this workflow might seem complicated, the logics behind it is quite intuitive which is observed in the following example. The part of a VD:

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

assuming for the sake of simplicity that results of the query

```
tnt:collection('/exercises//*.omdoc')//dc:creator/text()
```

are "Paul" and "John", will be substituted by:

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

VDs allow an arbitrary number of **tnt:xqinclude** elements in the VD Skel (not nested, though) with different XQueries. XQueries can be defined in 4 ways: in the attribute, in the child element as text, as a reference to outside defined queries (see next bullet) and as a reference to another file in a TNTBase repository that contains the implied query. It is worth mentioning that VD Specs may contain only references to skeletons in other VD Specs and differ just in queries or parameters. This approach becomes very handy when we want to leverage from the same skeleton but tweak another parts of a VD Spec, i.e. queries or parameters (see below for examples).

Queries Apart from being defined in **tnt:xqinclude** elements, XQueries can also be described in separate **tnt:query** elements, again as a text or as a reference to another file in the repository. Query should contain a name that serves as a link point from a VD Skel. There are no constraints on a query: it may reference *older* revisions of documents (that justifies a temporal aspect of TNTBase), *other VDs* or *auxiliary information* associated with documents, e.g. RDF (see [ZKR10] for more details). Such "external" query definition may be handy when we want to *override* the queries for a particular VD Skel. Getting back to our example, assume that we want to embed only statements of assignments in our VD preserving the common structure. We do not need to modify a VD Skel. Instead we create a new VD Spec that references the existing one with an overriding XQuery:

```
<tnt:virtualdocument xmlns:tnt="http://tntbase.mathweb.org/ns">
  <tnt:skeleton href="/basic-spec.xml"/>
  <tnt:query name="exercises.xq">
    for $t in $topics return
      tnt:collection(concat('/exercises/', $t, '/*.omdoc'))//exercise/statement
  </tnt:query>
</tnt:virtualdocument>
```

Here we assume that the full-fledged VD Skel can be found in the VD Spec under the path /basic-spec.xml. Thus we can *"inherit"* VD Skels recursively and override XQueries in any

combination that comprised quite a flexible mechanism to reuse existing VD Skels and queries and override only parts when needed. Also it is possible to create VD Specs whose skeletons reference queries that are not present in the same VD Spec. This feature is comparable to e.g. Java abstract classes, i.e. such a VD Spec cannot be used as such, but can be referenced from another VD Specs that defines the missing queries.

Parameters XQueries can reference variables that are not defined in the current context. Those can be externally defined in the **tnt:param** elements outside the query. This approach separates logics from the input. Similarly to queries, parameters can also be overridden or be absent in a particular VD Spec. In the latter case, VD Specs that inherit the current VD Spec should define absent parameters. In our example in Listing 1 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 VDs.

In this subsection we described the *VD Specs* — means to define the structure of VDs. In Figure 2 a VD Spec is denoted as the left bottom picture in the life cycle of VDs. But how do we *handle* and *consume* VD content?

3.2 VDs as TNTBase FS Entities and Their Materializing

In order to make VDs as a part of a TNTBase file system and expose them to users, one has to utilize the RESTful API of TNTBase [Z+10]. When creating a VD, a user has to provide a path and a name of a VD, a VD Spec and its revision which a VD will be linked to and, optionally, a set of parameters, analogously to those that a VD Spec has (thus, one can override VD Spec parameters or define new ones). Note that it is possible to associate a VD with a VD Spec of a *particular revision*, not only with the *HEAD* revision. Parameters associated with VD file system entities make VD Spec even more reusable. When retrieving content of a VD (i.e. the expanded version of a VD Spec), a user might also provide parameters that will override those defined in a VD Spec and in a VD itself. It might be very useful for dynamic alternation of a VD or during debugging. In Figure 2 the content of a VD is presented in the second picture at the bottom. We see how data are aggregated and mashed up with the static parts of a VD Spec.

The TNTBase RESTful interface also provides a related feature of dealing with VDs: we can turn the content of a VD into a regular file in a repository. We call such a process as ***VD materializing***. It is convenient when VD computation takes considerable amount of time whereas the content of a repository is changed rarely, or a user intends to fix the content and make it versioned: in our example when a user is satisfied with exercises list he got through a VD, he might want to make it persistent by putting it into a repository under certain path (i.e. by materializing a VD). If there is already a document under provided path, materializing results in a new revision of that file. Thus we can version even VDs with a possibility to roll back in case of necessity.

3.3 Querying VDs

The contents of a VD can be addressed in a query via the TNTBase XQuery extension function **tnt:vdoc**(`$path as xs:string`), where `$path` is a path of a VD in a TNTBase repository. Thus one may combine querying of usual repository files together with multiple VDs. It is

also possible to retrieve just the expanded version of a VD Spec (not the content of a VD which might be different due to additionally defined parameters). The XQuery extension function looks similar - tnt:vd−spec($path as xs:string), but instead of a VD path, we provide a path to a VD Spec. As an example, assume that we want to get a number of authors that contributed to exercises in a VD. The query will be simple:

```
count(tnt:vdoc('/path/to/vd')//dc:author)
```

3.4 VD Editing

One of the strongest features of VDs is that they can be edited and *committed* to TNTBase via the RESTful interface: changed parts of a VD that came from files in a repository will be transparently propagated back to the sources with repository history preservation, i.e. a new revision will appear in TNTBase. In Figure 2 on the bottom right picture we can see the final phase of a VD workflow: editing and submitting it back: the modified parts (marked with red) are populated back to their "home". All changes are performed in a single xSVN transaction and only those files will be part of it that were implicitly affected by VD editing.

In a VD we distinguish *static* parts (i.e. those that come from the VD Spec) from *generated* ones (i.e. those parts that are results of a particular VD Spec query). Currently, static parts are not editable in a VD – TNTBase will abort a commit, if they have been changed, however, static parts can be modified by changing VD Spec file directly in a repository. From the generated part, only the XML elements that come from the repository are *editable*; TNTBase annotates them with **tnt:doc** and **tnt:xpath** attributes that cache information about the element source needed for the commit. In our exercise example an editable part might look like:

```
<exercise topic="algebra" tnt:xpath="/omdoc[1]/theory[3]/exercise[2]"
                          tnt:doc="/exercises/algebra/isomorphic−sets.omdoc">
  ...
</exercise>
...
<exercise topic="algebra" tnt:xpath="/omdoc[1]/theory[2]/exercise[1]"
                          tnt:doc="/exercises/algebra/relations.omdoc">
  ...
</exercise>
```

A user may add attributes, text, comments, new elements or delete the old ones, but he is *not* allowed to modify **tnt:doc** and **tnt:xpath** attributes, otherwise TNTBase will abort committing.

So far we have considered only *editable* parts of the generated part. There could be, however, non-editable, generated nodes (e.g. dates or average of some values), which we call *constructed*. In some cases, we can even partially modify these and propagate changes back to a repository: If XQueries wrap some of DB XML elements into some other elements, making these constructed. For example, the query associated with some tnt:xqinclude element could be:

```
<author>{tnt:collection('/exercises//*.omdoc')//dc:creator}</author>
```

So the results might be:

```
<author>
 <dc:creator>Paul</dc:creator>
 <dc:creator>John</dc:creator>
</author>
```

In such cases we can make these elements editable via the XQuery *wrapper* function tnt:make−editable supplied by TNTBase for this purpose. This function adds **tnt:doc** and **tnt:xpath** attributes to the wrapped elements if possible (i.e. if the input sequence of elements are nodes in DB XML) to mark them as editable. So in our example the query should really be:

```
<author>{tnt:make−editable(tnt:collection('/exercises//∗.omdoc')//dc:creator)}</author>
```

This leads to the following result which parts are editable:

```
<author>
 <dc:creator tnt:doc='...' tnt:xpath='...'>Paul</dc:creator>
 <dc:creator tnt:doc='...' tnt:xpath='...'>John</dc:creator>
</author>
```

The editing approach has a number of natural limitations:

1. If VD content contains multiple results that are the same document node in a repository, then TNTBase will not allow committing this VD either, since in this case it is not clear which of modified nodes should be propagated to a source document. In future, this behavior might be changed so that e.g. the first or the last change of the same node wins and is sent to a repository.
2. If VD dynamic parts came from older revisions of repository files, then such parts can not be editable as well, since once revision is committed to a repository it becomes unchangeable.
3. From a dynamic part only XML elements[2] are editable. This limitation stem from the fact that e.g. text may be produced by concatenation of multiple strings inside some element and TNTBase will not be able to determine how to propagate changes back and there is no place to put source information inside a VD (the latter holds for all remained types of XML nodes as well). However, it is possible to work around this limitation by embedding XQueries that return those nodes together with their parent element.

Last but not least, TNTBase follows the "update-modify-commit-merge" cycle from the underlying version control system in this process, so if another user modified repository contents while we were editing a VD, submitting of the latter will fail, since our VD is "out-of-date" and we have to get the VD content again and merge our changes. This mechanism guarantees that we will not overwrite somebody else's changes.

3.5 VD Schema Validation

The obvious well-formedness constraints for a VD Spec can easily be expressed in a RelaxNG schema, which we supply as [ZK]. But schema-validity of the VD Spec does not ensure validity of the resulting VD, since it does not take the constraints of the target format into account. Fortunately, we can easily integrate the VD Spec schema with the target schema, if the latter meets (or is extended to meet) some modularity requirements. As an example we provide such a combination for OMDoc language in Listing 2.

Listing 2: A RelaxNG Schema for Virtual OMDoc Specifications

```
include "tnt−vd−spec.rnc" {
        skeleton.model  = grammar {include "omdoc/omdoc.rnc"
                                                {ss |= parent xq?}}
        return.internal.el      = grammar {include "omdoc/omdoc.rnc"
```

[2] i.e. that are of XML Schema type xs:element

```
5                                   {start= omdoc.class|CMP|FMP
                                     ss |= parent result . el}}}
```

Here we made use of the fact that the OMDoc schema has a hook (the ss schema macro) that allows replacement of elements in the declarations: all element declarations are of the form

```
CMP = (ss | element CMP {CMP.attribs & CMP.model})
```

In this situation, we only had to replace the content and attribute model of the **tnt:skeleton** element[3] with the OMDoc document model, which is upgraded to allow an **tnt:xqinclude** element in place of all "ss-replaceable" elements. The content model of the **tnt:return** element is instantiated to the model of OMDoc "ss-replaceable" fragments, updated to allow **tnt:result** element in place of "ss-replaceable" elements. When VD format-aware schema is ready, we can associate it with a VD Spec via tntbase:validate xSVN property (details in [ZKR10]), and thus ensure that TNTBase allows committing only those VD Specs which would always produce content that is valid against our format schema.

3.6 Implementation Details

VDs are realized[4] in XQuery with help of XQuery external functions written in Java. External function are used, for instance, for dynamic query execution from another XQuery (for expansion of a VD Spec), getting the revision information from a repository (to control that no other revisions have been committed while editing a VD) or committing changes under certain path (to propagate changes to original files one edited VD has been submitted back). In order to support VD editing workflow, there is a simple XQuery implementation of XML differencing, that

- controls that VD static parts were not modified,
- controls that **tnt:doc** and **tnt:path** attributes of editable parts were not modified,
- aggregates information about changed editable parts and groups them by source file,
- checks that there are no more than one editable part that corresponds to the same node in the same XML document.

4 Use Cases

In this section we discuss four real-world use cases of the VD technology. The discussion here is complemented with an evolving[5] TNTBase sandbox installation [Zho] that supplies Relax NG schemas and shows VD queries and VD Specs of our use cases in action.

4.1 Automated Exam Generation

This is a dogfood use case from our academic practice, and is (partially) used in day-to-day operation: The second author teaches a first-year, two-semester Introduction to Computer

[3] which allowed arbitrary elements that contain **tnt:xqinclude** queries

[4] Due to some problems in DB XML concerning multiple imported modules or support of XQuery external functions written in different languages, VD functionality cannot yet be fully integrated, we expect to have a fix for this by the conference.

[5] At the time of the submission, the sandbox is not fully operational yet, but we expect it to be the base of our system demos at the conference

Science a Jacobs university and – over the last six years – has accumulated a collection of about 1000 homework, quiz, and exam problems encoded into the XML-based OMDoc format [Koh]. 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 (and add to the corpus of well-tested problems), we assemble the exams from it semi-automatically with a VD 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 75 minutes or so, whereas finals are designed for 2 hours. Thus we also provide an exam duration as an input parameter for our *exam VD*. 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. Our XQuery for a VD Spec takes care about adjusting the timing closely to the provided limit. When VD content is generated, it can be rendered by utilizing XSLTs and developed in our JOMDoc library [JOM10] for rendering MathML[W3C07]. Everything is embedded into TNTBase, and once an exam VD is installed it is a matter of one click in the TNTBase web interface to get the unique human-readable exam sheet for the students.

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

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

Although a presented approach already meets our requirements, there are some issues that could be improved. 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). That will lead to a more complicated queries for a VD Spec, but is still feasible. Apart from generating exams, this use cases 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 complex ones. Some parameters in e.g. cookies may keep track of what exercises already appeared in the practice sheet, and a VD will never show them again. It could easily be done by providing dynamic parameters to a VD retrieval method as was described in the previous section.

4.2 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 like XQuery 1.0/1.1, XPath 1/2, XML 1.0/1.1., or even MathML1.0/1.0.1/2.0/2.0(2e)/3.0. They are encoded in XML format XMLSpec [XML09], so TNTBase and VDs apply. Usually some parts of specification remained the same, while other parts change between the version, 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 an *Diff VD* 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 a document XMLSpec documents upgraded with difference alternatives. Note that our XQuery-based XML-diff comes in handy here. This VD 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 Diff VD is also editable that allows a user to fix obvious bugs right on spot, without navigating to the source file. Once Diff VD is settled 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 VD 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 VD which mixes document parts and queries is beneficial. Moreover, there can be multiple Diff VDs for tracking (and editing) various aspects of the differences in the specifications. Such Diff VDs may even take over the role of conflict editors we currently have in version control aware IDEs.

4.3 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 (*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 (*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 VDs may naturally come into play as we have seen above.

A very simple application of VDs 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 [WM08] these aggreated 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, VDs make separate the conceptually issue of auto-aggregation and presentation, which allows to support workflows like previews/editing of aggregated document parts and materialization (e.g. of branches and tags) for archiving.

Another simple application of VDs in technical payload documents is in XML-based literate programming [Knu92], where program text is intermingled with its documentation and explanation in a single tocument. Here a VD 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 MathML3 Recommendation [ABC+09] from which we generate the MathML3 RelaxNG Schema. A VD would have considerably simplified this process.

We have already seen Diff VDs as examples of managment VDs for version management in the last section. But VDs 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 move on to other proofreading tasks.

4.4 Refactoring Ontologies

Finally, VDs can be used for refactoring OWL [SWM04] Ontologies that are written in XML Syntax, e.g. OWL 2 XML [MPPS09]. VDs becomes very handy when making changes to a small subsets of multiple large ontologies. In the first phase we can preview ontologies changes in a VD using XQuery transform functions. In the second phase, when we are satisfied with results we can materialize a VD thus obtaining a refactored ontology as a usual document in a repository. Refactorings that can be done using VDs include renaming entities, factoring out or merging modules, rewriting axioms, lowering expressivity or stripping axiom annotations. For more detailed information concerning ontology refactoring using VDs refer to [LZ10].

5 Conclusion & Further Work

In this paper, we have presented the concept of Virtual Documents and its prototypical realization in our TNTBase system. VDs integrate computational facilities into documents like JSP/PHP or TEX/LATEX, only that VDs use the versatile and XML-optimized XQuery processing as a computational process instead of relational database lookup (PHP) or general macro expansion in the latter case. 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 VDs are one way of introducing this to the XML world. We feel that we have just skimmed the practical possibilities induced by VDs in the use cases discussed in Section 4.

For instance, we envision that VDs 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 VD would contain an XQuery that takes specific to a particular user parameters like interested sections or ids 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 VD can satisfy needs of multiple users at the same time. Consider for instance the following situation: If the content collection contains information conceptual dependencies, then we can use VDs to generate *guided tours* [**?**], i.e. self-contained sub-documents introduce the necessary pre-requisites of a concept. As VDs allow to re-use the parametric XQueries that operationalize e.g. the topological sorting of concept 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 advaced e-learning systems like ActiveMath [**?**] can be externalized into VDs.

Note that the viability of virtual documents is intertwined with the targeted document formats in an interesting way as our discussion of validation in Section 3.5 shows. In [ZKR10] we have begun an exploration on a theoretical level; from our practical work reported in this paper it seems that more theoretical investigations are necessary. Note furthermore that our realization of VDs is not tied to the TNTBase system – even though version management

can profit from VDs, it is not a prerequisite; instead of an SVN commit we could just as well write to an XML database. In particular, as our implementation is based on XQuery in its core, it should be possible to port it to other XML databases if they supply a notion of a file system interface.

In our use cases, the ability to re-use XQueries[6] for different situations and over time has been a crucial ingredient for practical use of VDs. We therefore anticipate that common XQueries will be rolled into extensions[7] 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 we want to develop on top of our TNTBase system. One such tool is an editing framework for VDs. Note that this is non-trivial, since — like their underlying XML formats — VDs 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 like JOBAD [JOB08] could be extended in order to inject JavaScript into XHTML that marks up the editable parts of a transformed VD with help of auxiliary TNTBase attributes (like **tnt:doc** and **tnt:xpath**). Special markup will allow that framework to figure out what parts of a presentational document correspond to what parts of the sources from which VD content was comprised. The result could be that every "editable" part of rendered XHTML contains a small button or a link for editing pressing which results in some popup that shows an editable fragment of an XML document. Pressing submit button will modify the original content of a VD and commit it back to TNTBase. XML sources will be transparently patched that will lead to an updated XHTML version of a considered VD. Such an approach will allow a typical user to understand better the meaning of a VD (with help of human-oriented presentations) as well as provide interactive means for utilizing the powerful concept of VD editing.

References

[ABC+09] Ron Ausbrooks, Stephen Buswell, David Carlisle, Giorgi Chavchanidze, Stéphane Dalmas, Stan Devitt, Angel Diaz, Sam Dooley, Roger Hunter, Patrick Ion, Michael Kohlhase, Azzeddine Lazrek, Paul Libbrecht, Bruce Miller, Robert Miner, Murray Sargent, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) version 3.0. W3C Candidate Recommendation of 15 December 2009, World Wide Web Consortium, 2009.

[BCF+07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery: An XML Query Language. W3C recommendation, World Wide Web Consortium (W3C), January 2007. available at `http://www.w3.org/TR/xquery/`.

[Ber09a] Berkeley DB. available at `http://www.oracle.com/technology/products/berkeley-db/index.html`, seen January 2009.

[Ber09b] Berkeley DB XML. available at `http://www.oracle.com/database/berkeley-db/xml/index.html`, seen January 2009.

[6] which require specialized expertise and therefore constitute a significant investment

[7] Think e.g. of tableofcontents, references, or index elements that abbreviate a respective XQueries.

[CDF+08] Don Chamberlin, Michael Dyck, Daniela Florescu, Jim Melton, Jonathan Robie, and Jérôme Siméon. XQUpdate: XQuery Update Facility 1.0. W3C Candidate Recommendation, World Wide Web Consortium (W3C), seen February 2008.

[DKL+10] Catalin David, Michael Kohlhase, Christoph Lange, Florian Rabe, Nikita Zhiltsov, and Vyacheslav Zholudev. Publishing math lecture notes as linked data. In Lora Aroyo, Grigoris Antoniou, and Eero Hyvönen, editors, *ESWC*, Lecture Notes in Computer Science. Springer, June 2010. In press.

[JOB08] JOBAD framework – JavaScript API for OMDoc-based active documents. `http://jomdoc.omdoc.org/wiki/JOBAD`, 2008.

[JOM10] JOMDoc project — Java library for OMDoc documents. `http://jomdoc.omdoc.org`, 2010. seen Feb.

[JSR09] JSR 311: JAX-RS: The Java API for RESTful Web Services, seen April 2009. available at `https://jsr311.dev.java.net/nonav/releases/1.0/index.html`.

[Knu92] Donald E. Knuth. *Literate Programming*. The University of Chicago Press, 1992.

[Koh] Michael Kohlhase. OMDoc: An open markup format for mathematical documents (latest released version). Specification, `http://www.omdoc.org/pubs/spec.pdf`.

[Koh10] Michael Kohlhase. An open markup format for mathematical documents OMDoc [version 1.6 (pre-2.0)]. Draft Specification, 2010.

[LAT] Latin: Logic atlas and integrator. `https://trac.omdoc.org/latin/`.

[LZ10] Christoph Lange and Vyacheslav Zholudev. Previewing OWL changes and refactorings using a flexible XML database. In Mathieu d'Aquin, Alexander García Castro, Christoph Lange, and Kim Viljanen, editors, *submitted to 1st Workshop on Ontology Repositories and Editors*, number 464 in CEUR Workshop Proceedings, Hersonissos, Greece, May 2010.

[Mil10] Bruce Miller. `LaTeXML`: A LaTeX to XML converter. Web Manual at `http://dlmf.nist.gov/LaTeXML/`, seen March 2010.

[Mis10] Dimitar Misev. Integrating SUMO and OMDoc. Bachelor's thesis, Computer Science, Jacobs University, Bremen, 2010.

[MPPS09] Boris Motik, Bijan Parsia, and Peter F. Patel-Schneider. OWL 2 web ontology language: XML serialization. W3C recommendation, World Wide Web Consortium (W3C), 10 2009.

[Ope] OpenLink Software. OpenLink universal integration middleware – Virtuoso product family. web page at `http://virtuoso.openlinksw.com`.

[Rel] A schema language for XML. available at `http://www.relaxng.org/`.

[SVN08] Subversion, seen June 2008. available at `http://subversion.tigris.org/`.

[SWM04] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL web ontology language guide. W3C Recommendation, World Wide Web Consortium (W3C), February 2004.

[W3C07] W3C. Mathematical Markup Language (MathML) Version 3.0 (Third Edition). `http://www.w3.org/TR/MathML3/`, 2007. Seen November 2007.

[WM08] Norman Walsh and Leonard Muellner. *DocBook 5.0: The Definitive Guide*. O'Reilly, 2008.

[XML09] The XML Spec schema and stylesheets. `http://www.w3.org/2002/xmlspec/`, seen March 2009.

[Z+10] Vyacheslav Zholudev et al. TNTBase – restful api. `https://tntbase.org/wiki/restful`, 2010.

[Zho] Vyacheslav Zholudev. Sandbox for Balisage 2010 – Virtual Documents. `https://tntbase.org/wiki/balisage2010`.

[ZK] Vyacheslav Zholudev and Michael Kohlhase. The RelaxNG schema for vd skeletons. `https://svn.tntbase.org/repos/tntbase/trunk/DbXmlAccessLib/resources/tnt-vd-spec.rnc`.

[ZK09] Vyacheslav Zholudev and Michael Kohlhase. TNTBase: a versioned storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.

[ZKR10] Vyacheslav Zholudev, Michael Kohlhase, and Florian Rabe. A [insert xml format] database for [insert cool application]. In *Proceedings of XML Prague 2010*, 2010.