TNTBase: a Versioned Storage for XML

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

July 15, 2009

Abstract

Version control systems like CVS and Subversion have transformed collaboration workflows in software engineering and made possible the globally distributed project teams we know from the Open Source phenomenon. On the other hand, XML is coming of age as a basis for document formats, and even though XML as a text-based format is amenable to version control in principle, the fact that version control systems work on files makes difficult the integration of fragment access techniques like XPath, XQuery that are currently revolutionizing XML workflows.

In this paper we present the TNTBase system, an open-source versioned XML database obtained by integrating Berkeley DB XML into the Subversion Server. The system is intended as a basis for collaborative editing and sharing XML-based documents. It integrates versioning and fragment access needed for fine-granular document content management.

Contents

1 Introduction

With the rapid growth of computers and Internet resources the communication between humans became much more efficient. The number of electronic documents and the speed of communication are growing rapidly. We see the development of a deep web (web content stored in Databases) from which the surface Web (what we see in our browsers) is generated. With the merging of XML fragment access techniques (most notably URIs [BLFM98] and XPath [CD99, BBC$^+$07]) and database techniques and the ongoing development of XML-based document formats, we are seeing the beginnings of a deep web of XML documents, where surface documents are assembled, aggregated and mashed up from background information in XML databases by techniques like XQuery [XQu07], and document (fragment) collections are managed by XQuery Update [XQU08].

At the same time, the Web is constantly changing — it has been estimated that 20% of the surface Web changes daily and 30% monthly [CGM00, FMNW03]. While archiving services like the `Wayback Machine` try to get a grip on this for the surface level, we really need an infrastructure for managing changes in the XML-based deep web.

Unfortunately, support for this has been very frugal. Version Control systems like CVS and Subversion [SVN08] which have transformed collaboration workflows in software engineering are deeply text-based (wrt. diff/patch/merge) and do not integrate well with XML databases and XQuery. Some relational databases address temporal aspects [DDL02], but this does not seem to have counterparts in the XML database or XQuery world. Wikis provide simple versioning functionalities, but these are largely hand-crafted into each system's (relational) database design.

In this paper we present the TNTBase system, an open-source versioned XML database obtained by integrating Berkeley DB XML [Ber09b] into the Subversion Server [SVN08]. The system is intended as an enabling technology that provides a basis for future XML-based document management systems that support collaborative editing and sharing by integrating the enabling technologies of versioning and fragment access needed for fine-granular document content management. Our aim is to make possible workflows and globally distributed project teams as we know them from Open Source projects.

The TNTBase system is developed in the context of the OMDoc project (Open Mathematical Documents [OMD, Koh06]), an XML-based representation format for the structure of mathematical knowledge and communication. Correspondingly, the development requirements for the TNTBase come out of OMDoc-based applications and their storage needs. We are experimenting with a math search engine [KŞ06], a collaborative community-based reader panta rhei [pan], the semantic wiki SWiM [Lan08], the learning system for mathematics ActiveMath [Act08], and a system for the verification of statements about programs VeriFun [Ver08].

But TNTBase as described here is independent of all of these and has no specialization to mathematical content. This will be added at another layer, re-implementing an earlier system [FK00, KF01, FK06], but other XML-based systems could be supported as well, e.g. semantic Wikis like IkeWiKi [Sch06], KiWi [SEG$^+$09], eLearning Systems [CNX08, Tea06], scientific document archives, etc.

In the next section we will review the state of the art in versioning and XML databases, describing the two systems we combine and extend for TNTBase. In section 3 we an overview of a TNTBase architecture and interfaces it exposes. To make an every part of the architecture picture clear we will continue with describing the core of TNTBase — the XML-enabled repository in Section 4 and the Java accessory library in Section 5. Section 6 showcases an advanced feature of TNTBase: Virtual Files. Section 7 concludes the paper.

2 State of the Art

The TNTBase system is based on two widespread open-source systems: Subversion and Berkeley DB XML. We provide a short description of those aspects of the systems that are relevant to TNTBase and discuss what is missing for versioned XML-storage.

2.1 Subversion

Subversion (SVN) is one of the most popular open-source client-server version control systems. On a server side SVN maintains versions and history of documents and directories in a repository. Users work with such a repository by *checking out* to a local working space the directory tree (a working copy). This maintenance is performed by the SVN client utility. After a working copy is checked out users can perform various actions with it [CSFP04]: change, update from a repository or propagate changes back to a repository, changing properties of directories or files, merging different source trees, etc. The *update* command performs merging of a local working copy with the latest version in a repository. In case when automated merging is not solvable, a user has to edit conflicting files manually. Afterwards in order to propagate local changes back to a repository a user performs a *commit*. Using above mentioned commands comprises the typical workflow encountered by SVN users. We have covered only the basic concepts, but that is enough to get a rough conception of SVN. In Section 4 we will show that the TNTBase core is a substitution of an SVN server.

SVN is not aware of content inside a repository (apart from distinguishing binary and text files). For SVN users it does not make a difference whether they store text files, PDFs or XSLT stylesheets. In particular, SVN does not support native XML processing like XML databases. By XML-processing we mean possibilities to query XML-documents, index them in order to improve querying performance, benefit from XQuery Update facilities [XQU08] or utilize transactional mechanism in order to keep collection of XML documents consistent. Thus when we are talking about XML storing we should look at the XML-databases which is a subject of the next subsection.

Another limitation of SVN is that the smallest versioned entity in its repository is a file. But for some users it might be desirable to abstract away from the notion of files, and work with XML objects like a section in scientific papers in the DocBook format or theorems or proofs in mathematical documents. Roughly speaking, a user should be able to get away from the file metaphor (see [MK08] for further ideas).

2.2 Berkeley DB XML

Berkeley DB XML (DB XML) is an open-source, XML-native embedded database. Embeddedness means that it is distributed as a library with a number of API for various programming languages like C++, Java, Perl, Ruby and some others. This approach does not have an overhead by having surrounding environment like servlets or stand-alone servers. Also the embeddedness eliminates some database administration costs. DB XML is built on top of Berkeley DB [Ber09a] which is used by such applications as SVN (the consequences of this are discussed in the section 4), the RPM Package Manager [RPM09], the MySQL database [MyS08] and Postfix [Pos09], to name just a few of the most notable. Berkeley DB is an open source, embeddable database with zero administration; and DB XML inherits its advantages and features (e.g. portability, transactions, replications, easy deployment, etc.) from it. Naturally DB XML extends this with the typical XML-native database features: XQuery-based [XQu07] access to documents (with XQuery Update facilities support), support of transactions, preparsed queries, content-based indexing, scalability, recovery and locking mechanisms and the ability to work in multi-threaded and multi-process environments. Furthermore DB XML has established a reputation of being a scalable and very productive XML-native database that makes it a good choice to base the TNTBase system on.

But unfortunately DB XML does not support versioning which is becoming more and more important when managing collections of XML documents. Some of the products [Ipe09, Mar09, Ora09] on the XML-native databases market actually support versioning in a way, but this versioning has a bunch of limitations in comparison to ordinary version control systems like SVN or CVS, and moreover they all have a commercial license. On the other hand there is no popular version control systems which treat XML in a special way. This should be a goal of TNTBase as well.

3 The System Design and Interfaces

The TNTBase architecture is presented in Figure 1. We tried to keep it simple and understandable for readers by not showing irrelevant parts of the system. The core of TNTBase is xSVN (see Section 4). It is managed by Apache's `mod_dav_svn` module or accessed by DB XML Accessor (see Section 5) locally on the same machine. Apache's `mod_dav_svn` module exposes an HTTP interface exactly like it is done in SVN. Thereby a user of TNTBase is able to work with TNTBase repository exactly in the same way as with a normal SVN repository via HTTP protocol including Apache's SVN authentication via `authz` and `groups` files. The non-XML content can be managed as well in TNTBase, but only via discussed xSVN's HTTP interface.

DB XML Accessor is able to work with XML-content in an xSVN repository. Actually it works directly only with a part of it, namely with an xSVN container by utilizing DB XML API. All indispensable information needed for XML-specific tasks is incorporated in a DB XML container using additional documents or metadata fields of documents. SVNKitAdapter comes into play when the revision information needs to be accessed, and acts as a mediator between an xSVN repository and DB XML Accessor. And in turn when DB XML Accessor intends to create a new revision in a xSVN repository it also exploits SVNKitAdapter functionality. In Figure 1 note that SVNKitAdapter does not work directly with xSVN, but accesses it via HTTP as SVNKit [SVN07] can not access BDB-based repositories via the local protocol. But because we expose SVN HTTP access, this is not a problem.

DB XML Accessor realizes a number of useful features but is able to access an xSVN repository only locally. In order to exhibit all its functionality to the world, RESTful interface of TNTBase is provided for users. The full specification can be found at `https://trac.mathweb.org/tntbase/wiki/info`, but to get a rough idea what a user is able to do with it, see Sections 5 and 6 devoted to the DB XML Accessor features. We use the Jersey [Jer09] library to implement a RESTful interface in TNTBase. Jersey is a reference implementations of JAX-RS (JSR 311), the Java API for RESTful Web Services [JSR09] and has simplified our implementation considerably.

Apart from RESTful interface, TNTBase provides a test web-form that allows users to play with a subset of the TNTBase functionality before using RESTful style of communication. For simple testing of RESTful interfaces we would suggest the Firefox plugin which could be found at `https://addons.mozilla.org/en-US/firefox/addon/9780`. Also an XML-content browser is available online that shows the TNTBase file system content including virtual files. Unfortunately TNTBase now supports authentication only when accessing its SVN interface. The united authentication for all interfaces is a subject for future work.[1]

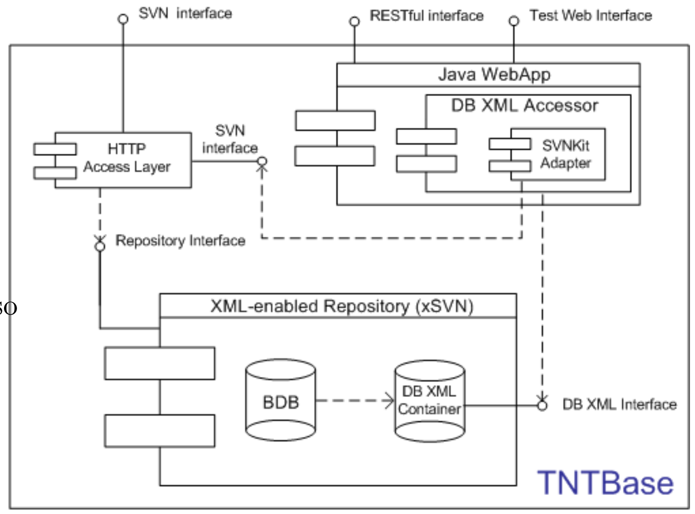

Figure 1: TNTBase architecture

Currently readers can access a test TNTBase system by two URLs: SVN interface at `https://alpha.tntbase.mathweb.org/repos/lectures/` and other interfaces at `http://alpha.tntbase.mathweb.org:8080/lectures/`. Additional information about TNTBase can be found on its TRAC page at `https://trac.mathweb.org/tntbase/`.

4 xSVN, an XML-enabled Repository

The architecture of xSVN and thus TNTBase is motivated by the following observation: Both the SVN server and the DB XML library are based on Berkeley DB (BDB). The SVN server uses it to store repository information[2], and DB XML uses for storing raw bytes of XML and for supporting consistency, recoverability and transactions. Moreover, transactions can be shared between BDB and DB XML. Let us look at the situation in more detail[3].

The SVN BDB-based file system uses multiple tables to store different repository information like information about locks, revisions, transactions, files, and directories, etc.. The two important tables for us are *representations* and *strings*. The *strings* table stores only raw bytes and one entry of this table could be any of these:

1. a file's contents or a delta[4] that reconstructs file contents
2. a directory entry list in special format called *skel* or a delta that reconstructs a directory entry list skel
3. a property list skel or a delta that reconstructs a property list skel

[1] see Ticket https://trac.mathweb.org/tntbase/ticket/3

[2] In fact SVN can also use a file-system based storage back end (SVN FS), but this does not affect TNTBase.

[3] The more comprehensive information could be found at `http://svn.collab.net/repos/svn/trunk/subversion/libsvn_fs_base/notes/structure` for the full story

[4] a difference between two versions of the same entity (directory entry lists, files, property lists) in a special format

From looking at a *strings* entry alone there is no way to tell what kind of data it represents; the SVN server uses the *representations* table for this. 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 — what it is a delta against. Note that the SVN server stores only the youngest revision (called the **head revision**) explicitly in the *strings* table. Other revisions of whatever entity (a file, a directory or a property list) are re-computed by recursively applying inverse deltas from the head revision.

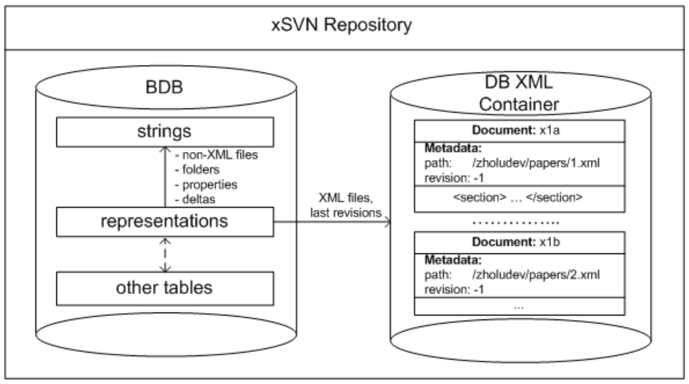

Figure 2: xSVN repository

To extend SVN to xSVN (an *XML-enabled repository*), we only need to subjoin the DB XML library to SVN and add a new type of entry in the *representations* table that points to the last version of that document in the DB XML *container* (see Figure 2). Containers are entities in DB XML that are used for storing XML documents in. Literally, a container is a file on disk that contains all the data associated with your documents, including metadata and indices. For every xSVN repository we use only one container located in the same folder as BDB tables, and therefore it allows us to share the same BDB environment exploited by an SVN back end.

From an end-user perspective there is no difference between SVN and xSVN: all the SVN commands are still available and have the same behavior. But for XML documents the internals are different. Assume that we commit a newly added XML file[5]. Its content does not go to the *strings* table, but instead a file is added to DB XML container with a name which is equal to the reference key stored in the also newly created *representations* entry of *DB XML full-text* type. When we commit a number of files and even one of the XML files is not well-formed then the commit fails and no data are added into an xSVN repository, which conforms to the notion of a transaction in SVN and DB XML. When we want to checkout or update a working copy, xSVN knows what files are stored in DB XML, and those files are read from a DB XML container. Another important thing is the scenario when we commit another version of an XML file. The older revision is deleted from DB XML, the newer revision is added to DB XML and a delta between these revisions are stored in the *strings* table. This delta has a normal text SVN format, and the SVN deltification algorithms have not been changed in xSVN (see Section **??**). 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 in 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 out of XML PDFs). And

[5]xSVN considers a file as an XML document if its extension is *.xml* or its *svn:mime-type* property is set to either *text/xml* or *application/xml*. This behavior can be easily adapted, for instance, by checking if a file starts with `<?xml`. Even now an SVN repository administrator can benefit from using automated property setting, i.e. associate certain file extensions with *text/xml svn:mime-type* property. For example, **.xslt* or **.xsd* would obtain *text/xml* mime-type on adding to a working copy and therefore will be treated as XML files for xSVN.

moreover we can add or commit XML and non-XML files in the same transaction.

As was mentioned above, xSVN deltification algorithms are inherited from normal SVN. The natural course of things for XML storage would be to substitute or extend these algorithms by XML-diff algorithms. We currently decided against this because SVN is a very complex system with differencing algorithms being an evidence of it. The more parts are subject for replacement or modification, the more efforts it requires and the less stable system becomes in comparison with the original well-tested one. Moreover, the text-based diff-algorithms are efficient, fast and reliable, nicely fit in with SVN architecture (to be precise, they are a part of it). Finally, it is not clear that there is any advantage to changing the deltification *on the server*. It is however clear that XML differencing brings great advantages *in the client* both in terms of smaller and less invasive deltas, and more informative conflict resolution strategies. But for transport in the server these "semantic differences" can be transformed into text-based diffs. If future research turns up advantages for supporting "semantic differences" in the server, we will integrate this into the xSVN server, otherwise we leave semantic differencing to the client layer integrated into TNTBase.

In conclusion: xSVN, as we presented it so far, offers a versioned XML storage, but without additional modules it is useless as the only difference from SVN is that it refuses to commit ill-formed XML documents.

5 The DB XML Accessor Library

So far we have introduced xSVN, an enhanced in our sense SVN, which stores the last revisions of XML files in DB XML instead of BDB. The next decision was to implement a Java library (DB XML Accessor) for internal usage which will serve as another brick to build the TNTBase system. So we have a DB XML container that contains all of the newest revisions of XML files, and we have a Java API for accessing this container. How do we proceed?

5.1 Querying XML Documents

We will start with a short description how querying is done in DB XML and in DB XML Accessor. As in nearly every XML-native database, the query language in DB XML is XQuery. To address the whole container in DB XML we use `collection('dbxml:/<container_name>')`. To access a particular document in a container DB XML uses `doc('dbxml:/<container_name>/<doc_name>')` syntax.

DB XML Accessor utilizes slightly extended and simplified syntax of accessing documents in a DB XML container. Since we have only one container in xSVN, to access all documents in a container just use `collection()`, to access a particular document use `doc(<path_to_doc>)`.

Here we should say something about how the latter query is transformed to DB XML syntax. As we mentioned in section 4, we use reference keys from the *representations* table as documents names in DB XML. There is another way to preserve a path and a document name in DB XML. To accomplish this DB XML document metadata are used. Each document in a container can have an arbitrary set of metadata fields of different types. This metadata could be also indexed by DB XML, which might improve performance of particular queries. So when xSVN adds a new XML document into a container, it also sets a document location in a repository, a document name and a full path of a document. For instance, if we have a document *paper.xml* in the */Balisage* folder, then the location in a repository would be */Balisage*, the document name — *paper.xml*, and the full path — */Balisage/paper.xml*. At first glance this information might seem redundant, especially taking into account that xSVN stores all of these in BDB tables. But by this approach we do not lose much except some storage space and writing performance when index of metadata should be updated. But we gain much more, now we are independent in DB XML Accessor from BDB tables, and each of the metadata fields can improve performance on particular queries which deal with documents paths.

Thanks to the mentioned above metadata fields, it is possible to access a subset of documents in a container. For this one should use `collection(<arbitrary_path>)` in DB XML Accessor. For example `collection(/doc*//test//paper??.xml)` would address all documents which names corresponds to the pattern `paper??.xml` (a '?' is just a wildcard) and they contain `test` directory in the path and the first directory of which starts with `doc`.

Also DB XML Accessor exposes methods for retrieving contents and paths of documents which are located at some arbitrary paths. Wildcards or '//', which stands for an arbitrary number of subfolders in a path, could also be used. All these queries would not be so efficient if we did not introduce a file system concept in xSVN container. Why did we have to introduce a file system? The answer is simple: DB XML does not have any hierarchical structure inside its containers. The next sections explains how we have introduced it.

5.2 File System in an xSVN Container

We already introduced the file path metadata fields in Section 5.1. Using them it is possible to reproduce the file system tree, but unfortunately it is not always efficiently. Assume that we want to find out what directories and files are located in a particular folder. For this we would have to execute a substring query on our file path metadata field. If a DB XML container contains a huge collection of documents then we could have a big delay while performing a seemingly simple task. The solution was to introduce ad-hoc XML documents in an xSVN container, one for each directory. We call such XML documents as *file system documents (FSDs)*. FSDs have a special name format: `tnt:<directory_path>`. Each of these FSDs contains a list of directory entries in XML format. For example for directory `/Balisage/papers` we might have the following FSD inside an xSVN container (its name is `tnt:/Balisage/papers/`):

Listing 1: A FSD in an xSVN container

```
<entries xmlns="http://tntbase.mathweb.org/ns">
  <dir name="sources"/>
  <dir name="references"/>
  <file name="paper_zholudev.xml"/>
  <file name="paper_kohlhase.xml"/>
  <vfile name="notations.vf" id="dbxml_54"/> <!-- will be explained later -->
</entries>
```

Now we can easily and efficiently find out about entries in the particular directory using XQuery. Here we should mention that such FSDs exist only for a folder which contain XML files or folders which contain XML files. Thereby we do not interfere with other content of an xSVN repository like text files or images. xSVN takes care about consistency in such FSDs, e.g. if a folder becomes empty after deletion of XML files, then the corresponding FSD is removed from a DB XML container and the folder is removed from the parent's folder entries and so on recursively. Also if we add some XML files to a newly created folder, then the file system structure is created recursively.

5.3 Write Access to TNTBase

So far we discussed only how to retrieve content and query an xSVN container by DB XML Accessor. But is it possible to write to xSVN using DB XML Accessor, or perform an XQUpdate query? The answer is positive. Then the next question arises, what would happen with revisions of XML files inside an xSVN repository? Shortly the answer is that the updated XML files will get a new revision in xSVN, then will be deltified, and a delta will be stored in BDB. Thus all history of modifications will be preserved. Let us discuss how we have accomplished that.

In DB XML Accessor we use the SVNKitAdapter library, which is based on the SVNKit — a Java library that re-implements the SVN client functionality. This allows us to work directly with an xSVN repository without a need to have a local working copy. In particular, SVNKit follows the SVN protocol to makes sure that no changes are lost on a commit; it forces an in-memory update and construct a delta between the local and the (updated) head revision. Only this delta is sent to a repository by SVNKit. But things get more complicated if we intend to modify an XML document by XQUpdate facilities. Then DB XML Accessor substitutes the original XQUpdate with a transform function (see [XQU08] for more details), which returns a modified document but does not modify a document internally in DB XML. Then this modified part is sent via SVNKitAdapter to xSVN in the usual way: SVNKit creates a new revision of a file and stores a delta against the previous version. Thus we can again retrieve a version of a file before executing XQUpdate. The xSVN log message would tell users how this change has occurred.

5.4 Querying Previous Revisions

Even though the xSVN container only stores the head revision of XML files we can query previous revisions of XML files: DB XML Accessor can *cache* XML files in the same xSVN container for the respective revision. Then we are able to query XML files exactly like we describe it in Section 5.1 but additionally providing a revision of interest. Note that only those files that have been cached before will be queried. Analogously we can remove a set of documents from a cache. Then they will not be queried. The advantage of this approach is that we choose manually the interesting subset of a revision thereby avoiding redundant filling of an xSVN container and eliminating unnecessary results. Also we are able to cache the single file unlike SVN when we are able to checkout or export only folders. Note that we can even cache the head revision, even though the head xSVN container already contains it. This can be useful when we intend to query against the documents of an exact revision: documents of the head revision can evolve, but cached documents remain the same.

All caching is mediated by SVNKitAdapter, which retrieves the necessary revisions from an xSVN repository. Then these revisions are added to a DB XML container with a special metadata field that denotes a revision number. This metadata field is also indexed, which improves performance on querying. All XML documents of the latest revision have a revision meta field equal to '-1'. This field allows us to distinguish different revisions when querying without loosing performance. In order to cache the latest revision, one should provide the exact number of it.

5.5 Caching Query Results

DB XML Accessor can cache query results in situations where a query incurs a large processing load, but the files that contribute to a query result change rarely. The user must simply pass a corresponding option to a query engine and receive a unique access handle with the computed result. Of course it is also possible to clean an xSVN container from cached results if they are not needed any longer or became obsolete. Internally, DB XML Accessor stores query results as separate documents. To distinguish them from e.g. FSDs introduced in Section 5.2, we introduce a *type* metadata field which is also applied for virtual files (see section 6).

6 Virtual Files

In this section we introduce a powerful concept — a *Virtual File (VF)*. A VF is a TNTBase file system entity which is a result of a particular XQuery expression, i.e. a VF is characterized by XQuery expression and a revision number this expression operates on. For instance if we create a VF with XQuery that returns the list of references from all scientific papers in a repository, then the content of a VF would be the list of references.

6.1 Creating a Virtual File and Getting Information about Virtual Files

A VF is a file system entity that records the following information:

1. an XQuery expression together with a list of namespace declarations which are used by it. The VF contents are determined by this XQUery expression. If XQuery provided is not valid, then TNTBase notifies the user and does not create the VF.

2. a revision number that a VF operates on. Note that if we did not cache any documents for that revision, then the content of a VF will be empty.

3. a description of what a VF does. This will simplify understanding for other users of VF intention. This field can be blank of course.

4. a VF path in a repository. It will be not allowed to create a VF if a file system entity already exists in the specified path. Even though it is possible to create a VF in folders which do not exist yet. In this case a directory structure will be created automatically.

For instance, if somebody is interested in all definitions from mathematical documents in a folder where a VF is being created, then (s)he can provide the following information to DB XML Accessor:

- XQuery: `collection(./*.omdoc)//ns:definitions`. together with the namespaces: `(ns, http://www.mathweb.org/omdoc)`. Note that the first '`.`' in the XQuery means the folder where a VF is being created.

- Revision number: `-1`. Stands for the head revision

- Description: `This VF returns all definitions from the current folder`

- Path: `/omdoc/theories/defs.vf`. A VF `defs.vf` will be created in the folder `/omdoc/theories`

After a VF has been created, one can easily retrieve its 'content', i.e. in our example all definitions in OMDoc documents in the `/omdoc/theories` folder. For the sake of example, a reader might also find useful Figure 3.

That is a typical creation procedure that is supported by DB XML Accessor. When a VF is created a new entity is added to a corresponding FSD. This entity is called *vfile* and also contains a name of a newly created DB XML document that encapsulates information about a VF. We call such a document as a **VF encapsulated document (VFED)**. To retrieve the content of a VF the corresponding FSD is checked for the VF. If a VF exists, then a name of a VFED is read. When we know the name of a VFED, then we are able to receive an XQuery expression from that document. As soon as we have an XQuery expression we can execute it and deliver results to a user. Namespace declarations which have been provided during a creation of a VF are used during XQuery execution and are stored in a VFED.

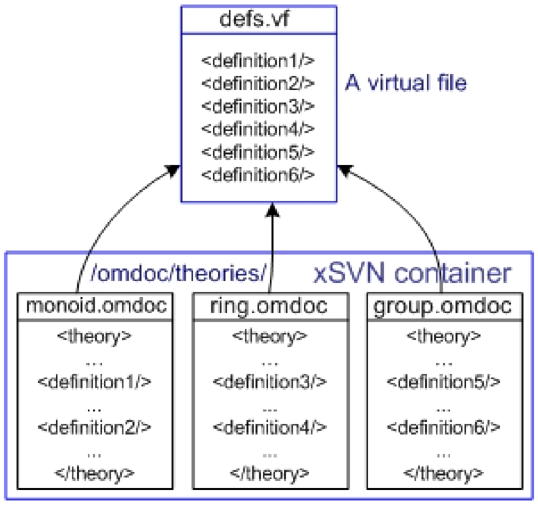

Figure 3: Definitions virtual file

VFEDs are tagged with the metadata field *type* discussed in Section 5.5. Therefore we can easily pick out only VFs and retrieve information about them in an xSVN container like their descriptions, revisions they operate on, their names, etc. Thus we are not get lost in the variety of VFs that users might have created.

6.2 Caching and Querying Virtual Files

To make VFs more like VIEWS in relational data bases, DB XML Accessor also allows them to be queried, but only if their content has been cached. This allows the user to specify which VFs participate in querying. Note that if XML files which form the content of a VF have been changed, the cache of a VF is not changed. This is a target for a future work[6]. Caching also might be useful when a user intends to receive a content of a VF quickly and is sure that the cache contains up-to-date data. This is especially worthwhile when an XQuery expression of a VF is computationally expensive.

When DB XML Accessor receives a command to cache a content of a VF (during creation of a VF, receiving VF's content or just via simple re-cache command), then the VF's content is stored in an xSVN container in the corresponding VFED. Results are wrapped in the special XML elements that are indexed. When querying VFs a user should use the same query syntax as (s)he uses for usual XML documents. That is possible because each VFED contains metadata fields for a VF path and its name. So for DB XML Accessor it does not make too much difference what is being queried: XML documents of the latest revision, cached documents of former revisions or VFs.

[6]see Ticket `https://trac.mathweb.org/tntbase/ticket/50`

6.3 Editing Virtual Files

We complete this section by introducing another operation that could be performed on VFs. We are talking about editing VFs, i.e. in some cases (which we will explain a bit later) it is possible to retrieve a VF for editing, modify it and submit changes back. Then the files from which a VF was formed will be modified in xSVN and will receive a new revision. Returning back to our example with a VF that contains definitions of mathematical objects, if we modify all definitions in this file, then all OMDoc files in the folder `/omdoc/theories` that contain definitions will be modified accordingly and committed to xSVN.

This approach has a number of limitations, some of them are quite obvious and straightforward:

- A VF we intend to modify should operate on the latest revision, since we can not change a particular revision in xSVN, because once committed a revision becomes persistent in a repository.
- We can edit only those VFs whose results are elements of some XML documents in DB XML, to be precise attributes or XML nodes. We rely on DB XML query engine to figure that out. If a result type is an XML node or attribute from DB XML point of view then we allow to edit such elements and show them in a list of results to be modified. But we can not edit pure text values which come from XML elements since text elements could be mixed with other XML elements, and when we retrieve such a text we lose the information before/after which nested element this text element has come from.
- The VF content is a set of results. Every result should be wrapped in a special XML element that contains the special information to allow DB XML Accessor to propagate changes back to original files. A user is only allowed to edit inside such elements, otherwise important information could be lost.

This allow us to get rid of a notion of files in a way and operate on the level of objects and version them, although internally in xSVN the minimal versioned entity is still a file. Let us provide an example of the VF that contains a list of creators in OMDoc files.

Listing 2: VFile with a list of creators

```
<?xml version="1.0" encoding="UTF-8"?>
<tnt:vfile name="defs.vf" mode="edit" xmlns:tnt="http://tntbase.mathweb.org/ns"
  xmlns="http://www.mathweb.org/omdoc" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <tnt:note>
    WARNING: do not edit 'results' elements, edit only within them!
    Otherwise TNTBase will not be able to version the corresponding original  files !
    Appending additional result elements may harm your TNTBase content.
    Additional elements under ' tnt:vfile '  other than ' tnt:result '  will  be ignored
  </tnt:note>
  <tnt:result file_path="/ecc.omdoc" element_path="/omdoc/metadata"
              element_name="dc:creator" element_type="element" id="1">
    <dc:creator role="trl">Michael Kohlhase</dc:creator>
  </tnt:result>
  <tnt:result file_path="/ecc.omdoc" element_path="/omdoc/metadata"
              element_name="dc:creator" element_type="element" id="2">
    <dc:creator role="ant">The OpenMath Society</dc:creator>
  </tnt:result>
  <tnt:result file_path="/omstd/arithmetics1.omdoc" element_path="/omdoc/symbol/metadata"
              element_name="dc:creator" element_type="element" id="3">
    <dc:creator role="ant">The TNTBase Society</dc:creator>
  </tnt:result>
</tnt:vfile>
```

This file is obtained for editing and as was discussed above contain wrappers with a set of attributes. Elements with the same name and path in the same documents are distinguished by the `id` attribute.

A couple of words how it is realized in DB XML Accessor. As we can see out of the example, every result of a VF is wrapped in a special XML element that contains the information about an original document, a path in the original document, a name of an element, a type of an element (an attribute or a node) and a unique id. All these items allow us to construct a unified XQUpdate expression for an every modified original document. So in our example, if we modify each element, then it means that we should propagate changes to two documents: `/ecc.omdoc` and `/omstd/arithmetics1.omdoc`. For the former one XQUpdate will contain two replacement statements, for the latter one — only one replacement statement. Then the technique described in Section 5.3 is applied.

7 Conclusion and Future Work

We have presented the TNTBase system, a versioned XML database system that can act as a storage solution for an XML-based deep web. The implementation effort has reached a state, where the system has enough features to be used in experimental applications. TNTBase may significantly ease implementation and experimentation of XML-based applications, as it allows us to offload the storage layer to a separate system. Moreover users which require only versioning functionality may use TNTBase as a version control system whereas more exigent users can experiment with additional features of the system.

The next development goals will be to stabilize the system further, to improve performance and extend it with special infrastructure for the OMDoc language. As an extended case study we want to develop TNTBase into an archive and content management system for scientific publications and semi-formal theories. A practical limitation that needs to be overcome on the way to this is the lack of a unified authentication and rights management subsystem.

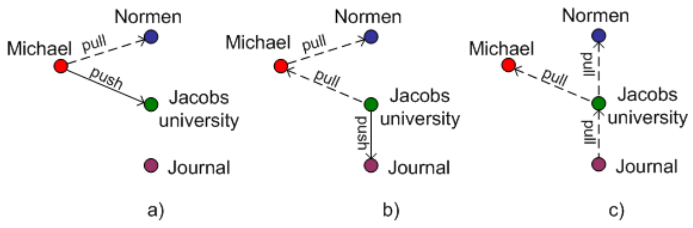

Figure 4: Distributed Scientific Publishing

In the near future, we want to study how the difference-based architecture inherent in version control systems can be extended to a distribution model. Consider for instance the situation in Figure 4: Michael started to work on his paper with future intentions to propagate it to Jacobs University. During the creation Normen wants to have a cache copy of a Michael's paper on his computer and look after the changes. From time to time Michael pushes his work to Jacobs University and the corresponding people at Jacobs checks the correctness of the paper. Then assume Figure 4b. When everything is done from Michael's side he wants to pass the rights for primary editing to university and only receive updates from it. Notice that Normen still depends on the Michael's updates. Now Jacobs University propagates its changes of the paper to some Journal and its stuff validates the correctness. Here is the same scenario as with Michael and Jacobs University in Figure 4a. Finally (see Figure 4c) Jacobs University passes the rights for original editing of the paper to a Journal (like Michael did with Jacobs) and Normen decides to switch the source of cached copy to Jacobs since he thinks that Jacobs contains more actual information. We assume that all the individuals and institutions in our examples are running TNTBase installations that store the relevant documents. Some instance of these are "originals", others are working copies that are updated from them and that commit to them. Crucially the TNTBase take over all the necessary caching and communication of differences to make the process transparent and effortless to the users. The preliminary idea is to implement a client library inside a TNTBase web application that will be taught to speak to other instances of TNTBase and receive information from them. This library would exploit the SVNKitAdapter module, which will be in charge of checking compatibility of documents versions and commit or update necessary paths in an xSVN repository.

Also our plans include the further work regarding virtual files. Some efficiency improvements should be done as well as more intelligent caching should be implemented. By "more intelligent" here we mean that the cache of virtual files should be updated automatically when the original files in a repository which form a VF content are changed. That would also mean the gain of performance since every time when we receive a content of a VF or query it, we can be sure that the cache is up-to-date and we do not need to regenerate it.

Finally, we plan to extend the XQuery family of languages with primitives for versioning to develop the full potential of the TNTBase system. The main operations here will be the propagation of changes, conflicts and non-interference judgments along semantic dependency relation; see [MK08] for first ideas. Currently much of the necessary content relations are language-dependent, but we will try to distill query and propagation primitives that can be implemented in the TNTBase system level.

References

[Act08] ActiveMath, seen September 2008. web page at `http://www.activemath.org/`.

[BBC+07] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. XML Path Language (XPath) Version 2.0. W3C recommendation, The World Wide Web Consortium, January 2007.

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

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

[BLFM98] Tim Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI), Generic Syntax. RFC 2717, Internet Engineering Task Force, 1998.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. W3C recommendation, The World Wide Web Consortium, November 1999.

[CGM00] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proc. of the 26th International Conference on Very Large Databases*, pages 200–209, 2000.

[CNX08] Connexions. web page at `http://cnx.org`, seen June 2008.

[CSFP04] Ben Collins-Sussman, Brian W. Fitzpatrick, and Michael Pilato. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[DDL02] C.J. Date, Hugh Darwen, and Nikos Lorentzos. *Temporal Data & the Relational Model*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2002.

[FK00] Andreas Franke and Michael Kohlhase. System description: MBase, an open mathematical knowledge base. In David McAllester, editor, *Automated Deduction – CADE-17*, number 1831 in LNAI, pages 455–459. Springer Verlag, 2000.

[FK06] Andreas Franke and Michael Kohlhase. MBase, an open mathematical knowledge base. In OMDoc – *An open markup format for mathematical documents [Version 1.2]* [Koh06], chapter 26.4.

[FMNW03] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet Wiener. A large-scale study of the evolution of web pages. In *WWW2003*. ACM Press, 2003.

[Ipe09] Ipedo XML Database, seen March 2009. available at `http://www.ipedo.com/html/ipedo_xml_db.html`.

[Jer09] Reference Implementation for building RESTful Web services, seen April 2009. available at `https://jersey.dev.java.net/`.

[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`.

[KF01] Michael Kohlhase and Andreas Franke. MBase: Representing knowledge and context for the integration of mathematical software systems. *Journal of Symbolic Computation; Special Issue on the Integration of Computer Algebra and Deduction Systems*, 32(4):365–402, 2001.

[Koh06] Michael Kohlhase. OMDoc – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, 2006.

[KŞ06] Michael Kohlhase and Ioan Şucan. A search engine for mathematical formulae. In Tetsuo Ida, Jacques Calmet, and Dongming Wang, editors, *Proceedings of Artificial Intelligence and Symbolic Computation, AISC'2006*, number 4120 in LNAI, pages 241–253. Springer Verlag, 2006.

[Lan08] Christoph Lange. SWiM: A semantic wiki for mathematical knowledge management. web page at `http://kwarc.info/projects/swim/`, seen October 2008.

[Mar09] MarkLogic Server, seen March 2009. available at `http://www.marklogic.com/product/marklogic-server.html`.

[MK08] Normen Müller and Michael Kohlhase. Fine-Granular Version Control & Redundancy Resolution. In Joachim Baumeister and Martin Atzmüller, editors, *Wissens- und Erfahrungsmanagement LWA (Lernen, Wissensentdeckung und Adaptivität) Conference Proceedings*, volume 448, 2008.

[MyS08] Mysql, seen June 2008. homepage at `http://www.mysql.com/`.

[OMD] OMDoc. web page at `http://omdoc.org`.

[Ora09] Oracle XML DB, seen April 2009. available at `http://www.oracle.com/technology/tech/xml/xmldb/index.html`.

[pan] The panta rhei Project. seen March 2009.

[Pos09] Postfix, seen May 2009. homepage at `http://www.postfix.org/`.

[RPM09] The rpm package manager, seen May 2009. homepage at `http://www.rpm.org/`.

[Sch06] Sebastian Schaffert. IkeWiki: A semantic wiki for collaborative knowledge management. In *1st International Workshop on Semantic Technologies in Collaborative Applications STICA 06, Manchester, UK*, June 2006.

[SEG+09] Sebastian Schaffert, Julia Eder, Szaby Grünwald, Thomas Kurz, Mihai Radulescu, Rolf Sint, and Stephanie Stroka. KiWi – a platform for semantic social software. In Christoph Lange, Sebastian Schaffert, Hala Skaf-Molli, and Max Völkel, editors, *Proceedings of the 4th Workshop on Semantic Wikis, European Semantic Web Conference 2009*, Hersonissos, Greece, June 2009. in press.

[SVN07] SVNKit - The only pure Java Subversion library in the world!, seen September 2007. available at `http://svnkit.com/`.

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

[Tea06] Connexions Team. Connexions: Sharing knowledge and building communities. White paper at `http://cnx.org/aboutus/publications/ConnexionsWhitePaper.pdf`, 2006.

[Ver08] VeriFun: A verifier for functional programs, seen February 2008. system homepage at `http://www.verifun.de/`.

[XQu07] XQuery: An XML Query Language, seen December 2007. available at `http://www.w3.org/TR/xquery/`.

[XQU08] XQUpdate: XQuery Update Facility 1.0, seen February 2008. available at `http://www.w3.org/TR/xquery-update-10/`.