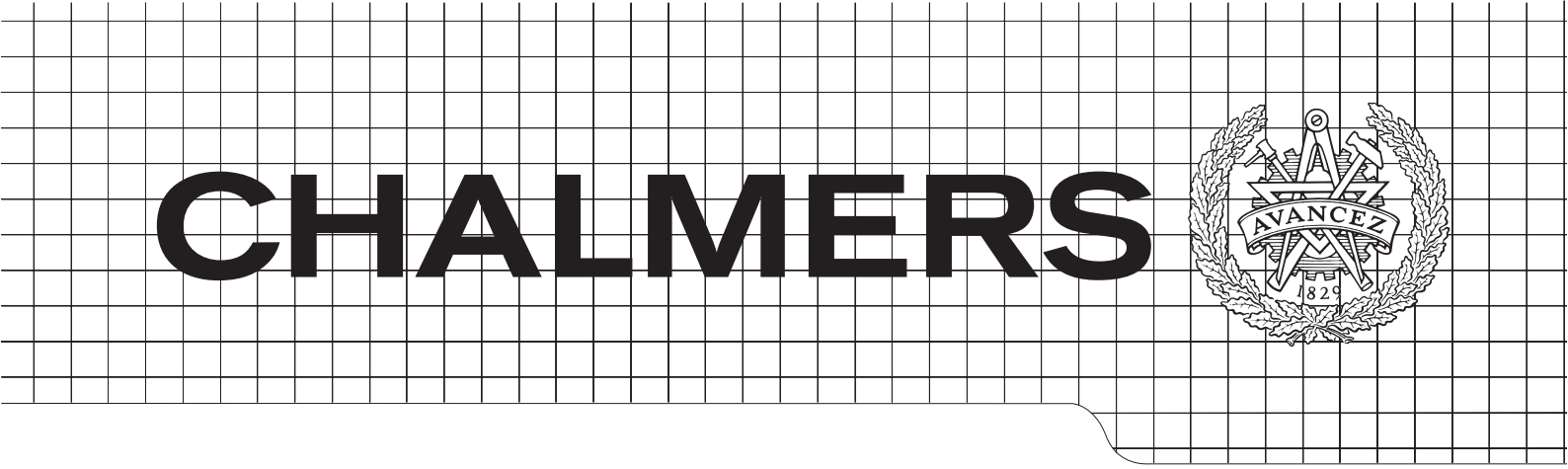


Efficient Temporal Queries in an XML-Based Content Management System

Master of Science Thesis

HUGO SVALLFORS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2013

Efficient Temporal Queries in an XML-Based Content Management System

H. SVALLFORS

Examiner: G. KEMP

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden September 2013

Abstract

The thesis concerns solving the following problem: How to efficiently search through multiple versions of the same XML file. A promising approach was found in the research field of temporal databases, and the works of F. Wang and C. Zaniolo in particular. They developed a pseudocode algorithm called the *XChronicler Algorithm* that given multiple revisions of an XML file, produces an XML *v-file*, an efficiently compressed, searchable history of an XML file that is itself an XML file. The algorithm is implemented and relevant parts of the implementation are discussed.

List of Figures

Contents

1

Introduction

The thesis concerns the following problem: How to efficiently search through multiple versions of the same XML file? This problem has been encountered by the company Repos AB, who have commissioned the author to solve the problem. In the process of doing so, this work solved the practical problem of the company, as well as finding an application for a mostly theoretical earlier work in the field of temporal databases [3].

1.1 Background

Repos AB is an IT-consultancy that primarily works within the field of client-side web development and XML. One of this company's primary products is ReposWeb, a content-management system (CMS) that stores documents as XML files inside the Subversion source control system. When combining these two technologies, the company feels that they attain a very flexible platform for storing structured text, that is very convenient to work with and tracks the history of the XML data stored.

The company does however find one significant flaw in this platform, and that is the ease of querying data stored within it. While Repos feels that plenty of off-the shelf solutions exist for querying a single XML file, they are trying to query multiple past versions of an XML file stored in Subversion. The company's current solution to doing so is to use the search library Solr, in conjunction with shell scripting to fetch the correct version(s) of the file to query. However, this leads to a situation were one must create a rather complex and fragile script for every query, leading to an unmaintainable amount of small scripts and too much workload for a mere one-off query.

In solution to this problem, the author proposed using the earlier work of C. Zaniolo and F. Wang [2] within the field of temporal databases [3]. This earlier work details an algorithm by which means a series of versions of an XML file can be combined into a *v-file*. This v-file is in itself an XML document, that contains every element from every version of the document used to generate it. However, the v-file only stores a series

of successive edit differences between each version, thus keeping storage use relatively manageable. Another strength of using this approach is that once such a v-file has been generated, performing a search of multiple past version of the document is as simple as performing a constrained search on the v-file. And for the problem of searching a single XML file, there are already as previously mentioned a great number of solutions.

1.2 Purpose

The purpose of this thesis is to implement a program that can create v-files from XML files stored in Subversion[1]. Having done that, the v-file(s) can then be queried by any number of commercial or open source search engines. This will solve the company's problem of querying an XML file's history, and also leave a significant degree of flexibility in their choice of search engine. Due to the v-file itself being XML, the v-files are relatively well supported by software tools. There are already a great deal of XML editors, evaluators, validators etc.

1.3 Outline

The rest of the thesis report is structured as follows: Section 2 contains a review of the literature regarding XML, XML Databases, Temporal Databases, and the paper this implementation is based on. In section 3, the approach to implementing the theory and some interesting algorithms and bugs that appeared are detailed. In section 4, the results of the implementation are shown via two test runs, to demonstrate that the software's functionality has been tested and to illustrate its workings. This is followed by a discussion of the results in section 5 and a conclusion in section 6.

[1] Also referred to as SVN.

2

Literature review

Before the solution eventually arrived at was proposed, a number of different related topics were researched. This section provides an overview of what was researched as well as a more detailed summary of the primary inspiration of the solution.

2.1 Apache Solr

As previously mentioned, Repos used Apache Solr [4] in their previous search solution. While this thesis does not directly use Solr, the system being replaced uses it, so it can be illustrative of the project purpose to describe Solr.

Solr is the most popular enterprise search engine, and is run as a open source project by the Apache Software Foundation. Solr can search through plain text for certain expressions, and highlight or filter hits based on user settings. It can also perform these searches over data in databases or many popular structured text formats, like XML in this case [5].

Solr is written in Java and can be run as a servlet within the Java servlet container framework. This enables easy deployment on Java servers such as Apache Tomcat [6] or Jetty [7]. Beyond being easily deployable on Java servers, Solr can also run as a standalone web service, and has REST-like [8] HTTP/XML and JSON [9] APIs. Combined with a framework for external configuration and plug-ins Solr is usable from almost all programming languages [5].

Repos finds Solr adequate for searching a single XML file, but the library is not intended to search through multiple versions of the same file. When trying to bend Solr to this purpose, one must use some kind of external script to fetch the versions to search trough, search them with Solr and then compare/collate the results. An extremely unwieldy process which has to be repeated for every query.

2.2 XML

XML (Extensible Markup Language) is a structured text markup language designed to be easily readable by both computers and humans. The syntax and semantics of XML are a gratis open standard which is curated by the W3C [10].

XML is most frequently used to exchange data over the network, although the language has also been used to encode everything from configuration files to office documents [11].

The standard places a strong emphasis on uniform serializing and parsing - XML is supposed to be readable and writable in the same way for any standard-compliant XML library. Being one of the most used data serialization formats, libraries and APIs that use XML in some capacity are very common and well established [10].

2.2.1 Terms & Definitions

Below are listed some commonly used terms used throughout this paper regarding XML and a short explanation.

- Character: XML is a text format. Almost the entire Unicode space is usable in an XML file [10].
- Markup and Content: The text content of an XML document can be categorized into either markup of content by the following rule: All the characters between a '<' and '>', or between an '&' and ';' are markup. Anything that is not markup is content [10].

  There are only two exceptions to this rule. The first being the CDATA section, where the text between the delimiters in a CDATA section is considered content, despite fulfilling the condition to be classified as markup. The second being that whitespace before and after the outermost element is classified as markup [10].
- Tag: A form of markup that begins with '<' and ends with '>'. Tags act as delimiters to elements. There are three kinds of tags in total [10]: Start-tags, like <body>, that start an element, end-tags, like </body> that end an element, and empty-element tags, like <img/>, that both start and end an element.
- Element: This is a component of the document which is delimited by tags as given above. Any characters between the start-tag and the end-tag are classified as content of the element. However, the content of one node can itself contain markup, including other elements, which are referred to as child elements. An example of an element is <example>An example element.</example> [10].
- Attribute: Another form of markup that consists of a name/value pair. Attributes can only be found in a start-tag or empty-element tag [10]. In fig. 2.1, we can see an example of an element with two attributes called src and alt:

Figure 2.1: Attribute example

```
<img src="foo.jpg" alt='A common placeholder name.'/>
```

- XML Tree: An XML document is a tree, with elements being nodes and attributes and text being leaf values [10].

2.3 XML Query Languages

The first topic of research was simply existing ways to query XML files, and in particular XML query languages. These languages share in common that they are domain-specific programming languages for extracting data out of an XML document.

One such language that was mentioned by the company itself was XPath [12]. XPath is a query language that can select elements or attributes of an XML document based on their value or position in the XML tree. Its syntax looks somewhat like a UNIX file path, the query "/foo/bar/baz" for example would select all elements with the tag name "baz" that were children of another element named "bar" that are in turn children of a "foo" element that is at the root of the document.

An XQuery expression, a.k.a a location path, is one or more location steps delimited by forward slash characters ('/'). There can be up to three components of a location step [12]:

- An Axis, which is a positional command relative to the node at the current point in the XPath. For example "foo" in "/foo/bar/baz" is an abbreviated form of the axis child::foo, which leads to a child of the current node with the name foo. Other axes, like @name for attributes or ancestor::name exist, where name is the node name.
- A predicate, which like other logical predicates select only elements that match the boolean expression. For example /foo/bar[@baz = "boo"] selects only those elements bar that have an attribute baz with the value "boo".
- A Node test, which gets a value from the node being tested. It can include node value, comment, text and the node itself among others. It can only occur at the end of the XPath or in predicates. For example /foo/text() retrieves the node text of the root element foo.

Another language is XQuery [1]. This can loosely be described as a "SQL for XML". It is much like SQL a declarative language that selects data, the difference being that this language selects it from XML trees instead of tables.

The syntax of XQuery is a superset of that of XPath as the language can use all XPath expressions [1]. However the XPaths are embedded in what is referred to as a

Figure 2.2: Example of FLWOR Expression from the XQuery Standard [1]

```
for $d in doc("depts.xml")//deptno
let $e := doc("emps.xml")//employee[deptno = $d]
where count($e) >= 10
order by avg($e/salary) descending
return
  <big-dept>
     { $d,
        <headcount>{count($e)}</headcount>,
        <avgsal>{avg($e/salary)}</avgsal>
     }
     </big-dept>
```

FLWOR expression (pronounced ”flower”). This is an acronym which stands for ”FOR LET WHERE ORDER BY RETURN”, after the keywords it may contain [1].

Much like one might expect from the name, FOR iterates over an expression and binds each value iterated over to a loop variable. ”FOR $var in /foo/bar” will iterate over every /foo/bar, binding each of them in turn to the variable ”var”, which can be used is the rest of the FLWOR expression.

LET simply binds a local variable, with the syntax ”LET $var := expr”.

WHERE works a lot like the similar keyword in SQL [1], in that it sets a predicate for the execution of the rest of the FLWOR expression. The syntax is simply ”WHERE expr”, where expr is a boolean. Both the loop variables and the let variables are in scope for this expression.

RETURN signifies the value that the expression to be returned if the execution reaches that point. The results may be ordered by ORDER BY, which simply sorts the returned values after another arbitrary expression.

To exemplify an entire FLWOR expression, we can use one from the W3C XQuery Standard [1], given in fig. 2.2, which given correct XML data will retrieve the departments that employ at least ten people, sort them in descending order by their salary average, and return a summary of the sub-departments of the greater department, along with their employee count and salary average.

2.4 XML Document Databases

There are several databases that store data in an XML format. Many of them allow you to query the XML data using one of the languages above, so one might say the relationship between these databases and the query languages is much like the one between e.g. MySQL and SQL.

A cursory review XML Document Databases turned up two of note. These are eXist-db [13] and BaseX [14]. Both allow you to store XML data and query it with XPath or XQuery through a local shell or a wide variety of web interfaces.

While neither of these would solve the problem given outright, they would eventually prove useful, as one can employ them to query the v-file, and make the v-file visible over the network if so desired.

2.5 Temporal Databases

A temporal database is a database with built-in support for tracking the changes in the stored data over time. Temporal databases stand in contrast to the norm of current databases, which store only facts which are believed to be true at the current time [3].

Commonly, this is implemented by storing any data along with a *transaction time* and/or a *valid time* [3]. Transaction time is represented as a pair of timestamps called *transaction-from* and *transaction-to* which denote when a fact is true in the *database*. Valid time is represented as a pair of timestamps called *valid-from* and *valid-to* which denote when a fact is true in the *real world*. Both of these kinds of temporal data save the lifetime of an object as a pair of timestamps that denote when it was created and deleted. The difference between the two types of timestamp is that valid time refers to time in the real world, whereas valid time refers to time in the database. E.g. if one had a temporal database that stored birth certificates, the valid time would refer to the person's actual real-life birth date, whereas the transaction time would refer to when the birth date was entered in the database.

Usually a temporal database will also enable you to query not only current data, but also past data or a range of past data. This is referred to as a *temporal query.* In most basic terms, a temporal query is a search through the temporal database constrained on it's transaction/valid time.

In this field a rather promising approach to dealing with XML files would eventually be found. In this approach, an efficient way to add transaction time to an XML file is detailed.

2.6 XChronicler

In the process of researching temporal databases that dealt with XML, a paper was found titled "Temporal queries and version management in XML-based document archives" [2]. In this paper, the authors detail a way to compress the history of an XML document to a single XML document that contains all data from all versions of the document. The algorithm that does this is called the *XChronicler* algorithm, and the XML file it produces is called a *v-file*.

The interesting part of this approach is how one avoids the size of the v-file utterly exploding beyond management. The way this is done is that elements that are constant between versions are not stored twice in the v-file. Instead, elements that can be identified as equal are only stored once, and they are annotated with their lifetime information:

Figure 2.3: Example File 1

```
<foo attr="attr">
  <bar/>
</foo>
```

Figure 2.4: Example File 2

```
<foo attr="attr">
  <baz/>
</foo>
```

The timestamp at which the element was created and which it was deleted (if applicable). This corresponds to the transaction time and transaction-from/to concept of other temporal databases. The authors refer to these elements with life cycle information as *tagged nodes*.

So for example, if one is creating a v-file from two versions of a single file given by fig. 2.3 and 2.4 the algorithm would convert them to the v-file in fig. 2.5.

The ”vstart” and ”vend” of each element signifies the version of the file at which the element was created and deleted respectively. The special value ”NOW” at the ”vend” of certain elements indicates that it has not yet been deleted, and is still alive in the latest version of the document.

The astute reader may already have noticed that the ”attr” attribute present in both versions of the file was converted into a child node of the same name. This is because when the v-file is updated with new changes, the updates are applied to a node by setting the ”vend” of the element in question to the current version, and then appending a new tagged node for the new version of the element. This ensures that the v-file really does contain all data from all versions and that the old version is not overwritten by the new. However, if one were to update an attribute of the root element of a document, the entire

Figure 2.5: V-File Generated from merging Example File 1 and 2

```
<foo vstart=0 vend="NOW">
  <attr isAttr="yes" vstart=0 vend="NOW">attr</attr>
  <bar vstart=0 vend=1/>
  <baz vstart=1 vend="NOW"/>
</foo>
```

Figure 2.6: XChronicler algorithm, as given in quoted paper [2]

1. Collect the timestamps of snapshot versions: $T_1; T_2; \ldots; T_n$;
2. Normalize snapshot version 1 as document X_1 by mapping attributes of elements as sub-elements, and converting elements into tagged nodes;
3. Timestamp document X_1 as V-Document V_1 by adding vstart and vend attributes;
4. $i \leftarrow 1$
5. while $i < n$
   (a) $i \leftarrow i + 1$
   (b) Normalize snapshot version i as document X_i ;
   (c) Compute the structured diff between document X_{i-1} and X_i and generate $Diff_{i-1,i}$;
   (d) Construct document V_i by applying update actions in $Diff_{i-1,i}$ on V_{i-1} ;

document would have to be duplicated!

To solve this efficiency problem, attributes are converted to tagged elements instead. Then the attribute can have life cycle information separate from its parent node, and any update to the attribute will not cause a drastic expansion in the v-file size. Child elements of the tagged node that encode attributes have the attribute ”isAttr=”YES””, to keep the distinction between elements and attributes.

Now that the workings of the XChronicler algorithm have been illustrated, we can show the algorithm itself. It is listed in fig. 2.6:

What the authors of XChronicler here refer to as a structured diff is an edit difference between two XML documents that is expressed not in terms of lines changed (like in a normal text diff) but in terms of XML elements added/removed/changed. The word ”normalize” above refers to the process of creating a v-file from the first version of a document, not to be confused with the XML standard’s definition of normalization.

The XChronicler algorithm provided an interesting approach to the problem of searching through an XML file’s history. By converting the history of an XML document into a v-file, this rather complicated problem suddenly turns into the much simpler, already solved problem of searching a single XML file. One can simply generate the v-file from the SVN history of an XML file, and then query it using XPath or XQuery. And so it was decided to employ a new implementation of XChronicler to solve the problem Repos AB had with their document database.

2.7 Related Work

Another paper roughly in the same field is "A Comparative Study of Version Management Schemes for XML Documents" [15]. This paper is a compendium of ways to efficiently store the history of an XML document. While this is very closely related to the XChronicler paper, it is not primarily focused on querying the history of the XML document.

Another similar work is "ArchIS: an XML-based approach to transaction-time temporal database systems" [16]. Rather than implementing a temporal database which stores XML, this paper implements a general temporal database using XML files.

Another related work is found in "Efficient Change Management of XML Documents" [17]. Much like [15] it is concerned with efficient ways of storing edit differences between XML documents, and reconstructing the original documents from these differences, but not to a similar degree querying that information.

3

Method

It was decided early in the project to implement the v-file generator in Java, due to Javas extensive libraries for parsing and modifying XML. This directly led to the project using SVNKit [18], as it is the only Java-accessible way of calling SVN. Another very useful library used by the project is XMLUnit [19]. XMLUnit is primarily a unit testing framework for XML files, but it also contains code for creating structured diffs between two XML files.

The largest parts of the pseudocode algorithm that are not specified in the paper [2] are: Input/Output for the v-file and document versions, and the precise way in which you attain a structured diff. The subject of this section is how these and other problems of implementing the pseudocode algorithm where solved.

3.1 Scope Changes

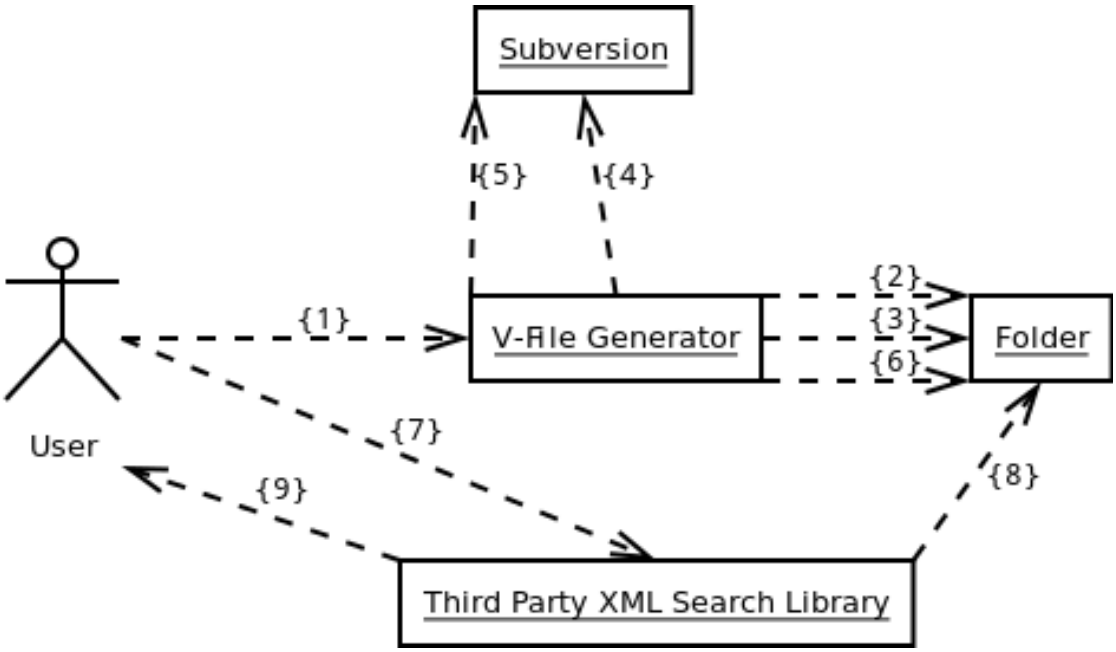
The project changed scope and goal numerous times throughout. This was in hindsight to be expected, as this was a very exploratory project and new territory for both the customer and the author.

In its first iteration, the customer had ordered an XML query language that had domain specific abilities to query SVN and run queries over multiple files. Upon discovery of XQuery, and that this software essentially already existed, it was decided to refocus the thesis towards being more of an XML search engine.

The second iteration was an XML Search engine. It would be a web server that had a restful web interface (e.g. received queries over HTTP URLs and sends responses as JSON). It would run queries against a v-file that it held updated to a given SVN repository. It was at this stage of the work that the XChronicler paper was found.

When this approach was described to the customer, they realized they could rely merely on the v-file itself and plug that into existing servers instead of commissioning a new one. Upon request by the customer, the server was scrapped, and just the v-file

Figure 3.1: Use Case Diagram



generation remained. The project would then just generate the v-file on being run, and searching would be handled by third party software.

3.2 Illustration of Use

This implementation of XChronicler is started as a command line application with three or more parameters: A SVN repository from which to fetch files, a local folder in which to place the v-files, and paths in the SVN repository to create v-files for. The local folder will be created if it is not already present, and then for every given path, the program will fetch the SVN repository log for that file. It will then fetch the versions given in the change log, and perform the XChronicler algorithm on them using XMLUnit to compute the structured diff. This is of course an extremely high-level overview of how it works, and there are several interesting details or issues in the implementation to discuss.

At a high level, the work flow for using the v-file generator is given in figure 3.1. Explanations of the actions are listed here.

1. The user starts the v-file generator. The generator is started from the command line, and is given a local folder to store v-files in, the URL to a remote SVN repository, and one or more file paths in the repository to generate v-files for.

2. The v-file generator checks if the folder exists, and if it already contains a file of the same relative path as the one being use to generate a v-file. If they do not exist, they will be created.

3. If the given folder and file *do* exist, they are interpreted as being a v-file generated from an earlier run of the v-file generator. The file will be parsed as a v-file, and updating can be resumed from the version saved in that v-file.

4. The generator will fetch the edit history of the file being used as base for the v-file. This is basically a list of versions at which the file changed. If an earlier v-file was detected, only the edit history since the last update will be fetched.

5. After that, the generator will fetch the text content of each changed version of the file, and diff them with each other as given by the XChronicler algorithm in fig. 2.6 to create a v-file.

6. After the update process is done, the v-file will be committed to disk, in a file with the same relative file path as in the repository, but stored in the given local folder.

7. When the v-file is generated, the user can then query it using an arbitrary third party XML search library. In this case, BaseX and XPath where used, but many alternatives exist. The user will provide the file path of the v-file and a query to run over it.

8. The third party search library will then run the query on the v-file.

9. The user gets the result of the query.

3.3 Class List

The classes in fig. 3.2 are used to implement the v-file generation. The responsibilities of some classes is briefly touched upon here. In section 3.4 a selection of the methods listed in fig. 3.2 will be discussed in detail.

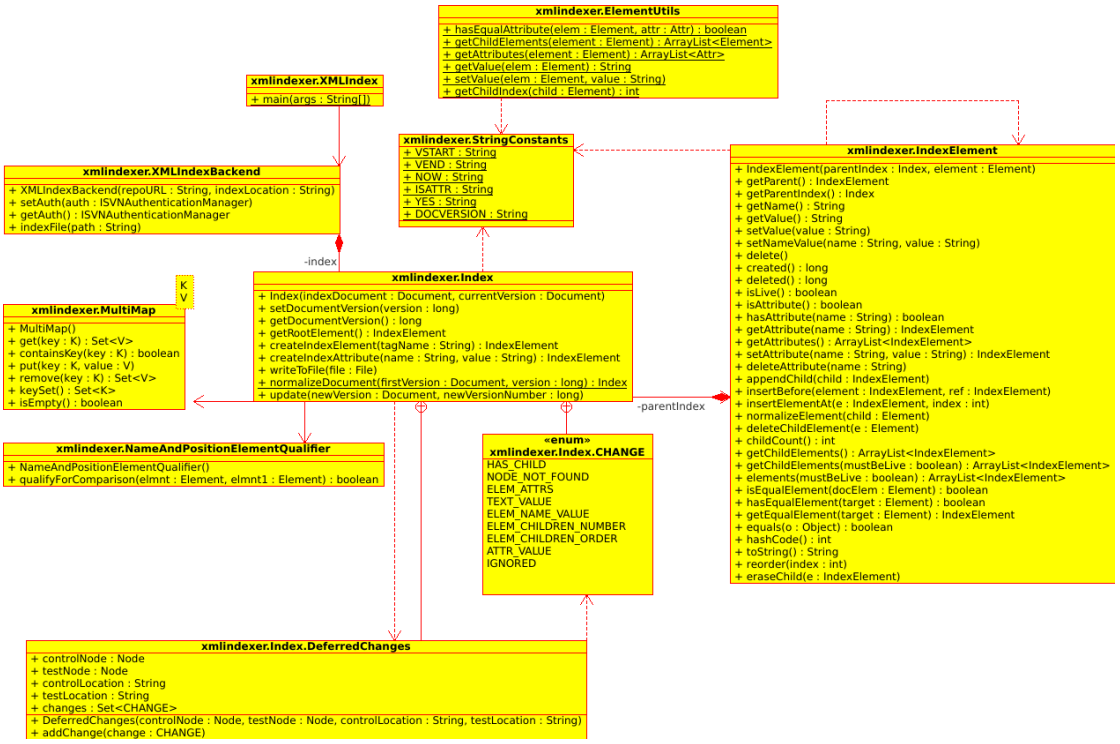
- XMLIndex: Front-end for the application. Contains main method and argument parsing.

- XMLIndexBackend: Responsible for fetching the log of a file from SVN, fetching files from SVN and parsing them as XML. Uses Index to create the actual v-file and update it.

- Index: Class that contains a single v-file XML document. Aggregates two XML documents: The v-file and the latest version of the file the v-file is derived from. Is updated by giving it a new document, which will then be differentiated against the old and the list of differences will be applied. Creates IndexElements.

- IndexElement: Corresponds to a single tagged element. Has methods to extract/-modify VSTART, VEND, and any value/children of the tagged node.

Figure 3.2: Class Diagram



3.4 A Selection of Methods

In this section, some of the non-trivial methods will be explained in detail, along with some of the decisions and bugs which explains why they were implemented this way.

Immediately when called, the XMLIndex.main (fig. 3.3) will create a new IndexBackend using the first two arguments at line 16 (that are parsed as an URL to an SVN repository and a local folder to put the v-file(s) in respectively). Then any subsequent arguments, of which there needs to be at least one, are parsed as file paths in the SVN repository to generate v-files from, and indexFile is invoked on them in (line 17-19).

Initially, indexFile (fig. 3.4) will create a path in the given folder that corresponds to the path it's been given to base the v-file on. (Line 18-19) E.g. if it is told to generate a v-file of the file http://url.com/foo/bar and put the result of doing so in the local file foldr/, it will first check if foldr/foo/bar already exists (at line 20). If it does, that file will be parsed as if it is an v-file, and the v-file generation will resume from the docVersion given in that v-file (line 33-40).

The v-file generation will proceed by, in line 20-40, fetching the change log of the file being used as base for the v-file. The only interesting information in the log for this purpose is that it contains the revisions at which the file was changed. If resuming from an already present v-file (line 33-40), we will fetch the changes from version docVersion + 1 onwards, and if creating a new one (line 21-32) we will do the same from version 0

Figure 3.3: XMLIndex.main

```
1  /**
   * @param args The commandline arguments.
3  * The first elements of args must be the URL to the SVN server,
   * the second is the location to build the index and
5  * any subsequent ones are files to be indexed.
   * @throws IllegalArgumentException If there are less than three arguments.
7  * @throws SVNException If the SVN repository could not be read.
   * @throws IOException If the index file or index folder could not be read/written
       .
9  * @throws SAXException If an XML document could not be parsed.
   */
11 public static void main(String[] args)
           throws SVNException, IOException, SAXException {
13     if (args.length < 3) {
           throw new IllegalArgumentException("Must have at least 3 arguments.");
15     }
       XMLIndexBackend idx = new XMLIndexBackend(args[0], args[1]);
17     for (int i = 2; i < args.length; i++) {
           idx.indexFile(args[i]);
19     }
   }
```

onwards.

The for-statement at the end of indexFile (line 43-54) roughly corresponds to the XChronicler algorithm that was given in pseudocode in fig. 2.6, but this is obscured somewhat by the diff generation and application both occurring in the Index method update.

Broadly speaking, what update (fig. 3.5) does is that it receives a new version of the document it is basing the v-file on. It will then diff the document to the last version (at line 9-10). This will generate a set of Difference classes, that are iterated through by scheduleChange (line 17-19), which classifies the changes into one of three maps created in this method. These maps are changeMap, newNodeMap, and reorderMap (declared at line 12-14). Basically all three map the nodes that are to be changed in the Index to changes to be performed.

One might ask why there are three of these maps as opposed to just saving all changes in changeMap. The reason for this is that the changes in the other two need to be performed in a different order, and so are separated out by this method. Reordermap for example, stores elements that have been moved relative its other siblings. However, that same node's parent may have added more siblings with this update, so node reorderings need to be performed after adding new child nodes in order to reliably work. Changes which are not particularly sensitive to the update order are left in changeMap.

After these maps have been generated, the changeMap will be given to updateIndex-Element (line 24), which applies them. After most of the changes have been applied, update will call a number of other methods which are explained in further detail below.

A relatively simple method, updateIndexElement(fig. 3.6) does a recursive-bottom-up update of the XML tree. It will first update the children of this node (line 13-15),

Figure 3.4: XMLIndexBackend.indexFile

```
/**
 * Fetches a single file from the SVN repository, given by path,
 * and creates a similarly named file in the index folder, containing
 * a v-file of that file's history. If such a file already exists,
 * it will be parsed as an index, and indexation will resume
 * from that indexes docVersion.
 * @param The path to index.
 * @throws SVNException If the path in question does not exist.
 * @throws IOException If the index could not be written to file.
 * @throws SAXException If either the index file or documents in the
 * SVN repo are not well-formed XML.
 */
public void indexFile(String path)
        throws SVNException, IOException, SAXException {
    System.out.println("Indexing " + path);
    ArrayList<SVNFileRevision> fileRevisions = new ArrayList<>();

    File indexFile =
            FileSystems.getDefault().getPath(indexLocation, path).toFile();
    if (!indexFile.exists()) {
        // Create new index file from first version.
        System.out.print("Fetching repository history...");
        repo.getFileRevisions(path, fileRevisions, 0, repo.getLatestRevision());
        long firstRev = fileRevisions.get(0).getRevision();
        Document doc = fetchDocumentVersion(path, firstRev);
        System.out.println("done.");
        System.out.print("Normalizing first version of " + path + "...");
        index = Index.normalizeDocument(doc, firstRev);
        fileRevisions.remove(0);
        index.writeToFile(indexFile);
        System.out.println("done.");
    } else {
        System.out.println("Parsing index at " + path + "...");
        index = parseIndex(path, indexFile);
        long lastRevision = index.getDocumentVersion();
        System.out.println("Parsed version " + lastRevision + " of index.");
        System.out.print("Fetching repository history...");
        repo.getFileRevisions(path, fileRevisions, lastRevision + 1,
                repo.getLatestRevision());
        System.out.println("done.");
    }

    for (SVNFileRevision sfr : fileRevisions) {
        long newRevision = sfr.getRevision();
        if (newRevision <= index.getDocumentVersion()) {
            continue;
        }
        System.out.print("Indexing version " + newRevision
                + " of " + path + "...");
        Document newDocument = fetchDocumentVersion(path, newRevision);
        index.update(newDocument, newRevision);
        index.writeToFile(indexFile);
        System.out.println("done.");
    }

    index.writeToFile(indexFile);
    System.out.println("Finished indexing " + path + "!");
}
```

Figure 3.5: Index.update

```
/**
 * Diffs the given document with the last version of it, and applies
 * the changes to the index.
 * @param newVersion The new version of the document.
 * @param newVersionNumber The version number of the document in SVN.
 */
public void update(Document newVersion, long newVersionNumber)
        throws IOException {
    DetailedDiff diff = new DetailedDiff(
            new Diff(currentVersion, newVersion));
    diff.overrideElementQualifier(new NameAndPositionElementQualifier());
    Map<IndexElement, DeferredChanges> changeMap = new LinkedHashMap<>();
    Map<IndexElement, DeferredChanges> reorderMap = new LinkedHashMap<>();
    MultiMap<String, Element> newNodeMap = new MultiMap<>();
    List<Difference> differences = diff.getAllDifferences();

    for (Difference d : differences) {
        scheduleChange(changeMap, reorderMap, newNodeMap, d);
    }

    currentVersion = newVersion;
    this.setDocumentVersion(newVersionNumber);

    updateIndexElement(changeMap, newNodeMap, this.getRootElement());
    addOrphanNodes(newNodeMap);
    reorderNodes(reorderMap);
    cleanDocument();
    assertEquals();
}
```

and then look up the node in the provided tables (line 16-19). If it is contained there, it will retrieve the updates to apply, and call sub-procedures to implement them (20-47), which were felt to be too trivial to list here.

The method addOrphanNodes(fig. 3.7) was added to resolve a rare edge case. In most cases, when new node is added to the Index, this will cause the parent's child list to be a bit longer, and so XMLUnit will detect a difference in child node list length. Differences of that type are applied in the previous method, and any new children of that node are added there.

So in most cases, new nodes are added in the previous method. However, there was an elusive bug with that approach which occurs in the following circumstances:

1. There exist in the old and new document a node with a single child.

2. The child node changes tag name.

We can see an example of two versions that fulfill these criteria in fig. 3.8 and 3.9.

When these circumstances hold, the old and new child will not match according to XMLUnit. As such XMLUnit will consider the old child node to have been deleted and the new node to have been added. *However, the parent child node list length will be one in both cases.* As such only the old node will be deleted, and the new will not be added.

Figure 3.6: Index.updateIndexElement

```
1  /**
   * Does a bottom-up update of the XML index.
3  * @param changeMap A map of IndexElements to changes to be performed
   * on that node.
5  * @param newNodeMap A map from XPath location of an element
   * to the new nodes that are to be added there.
7  * @param element The element being updated.
   */
9  private void updateIndexElement(
           Map<IndexElement, DeferredChanges> changeMap,
11         MultiMap<String, Element> newNodeMap,
           IndexElement element) {
13     for (IndexElement child : element.elements(true)) {
           updateIndexElement(changeMap, newNodeMap, child);
15     }
       if (!changeMap.containsKey(element)) {
17         return;
       }
19     DeferredChanges d = changeMap.get(element);
       for (CHANGE change : d.changes) {
21         switch (change) {
               case NODE_NOT_FOUND:
23                 element.delete();
                   return;
25             case ELEM_CHILDREN_NUMBER:
                   updateElementChildren(newNodeMap, element, d.testLocation);
27                 break;
               case HAS_CHILD:
29                 updateElementChild(element, d.controlNode, d.testNode);
                   break;
31             case TEXT_VALUE:
                   updateTextValue(element, d.testNode);
33                 break;
               case ELEM_NAME_VALUE:
35                 updateElementNameValue(element, d.testNode);
                   break;
37             case ATTR_VALUE:
                   updateAttributeValue(element, d.testNode);
39                 break;
               case ELEM_ATTRS:
41                 updateElementAttrs(element, d.controlNode, d.testNode);
                   break;
43             default:
                   throw new UnsupportedOperationException();
45         }
       }
47 }
```

Figure 3.7: Index.addOrphanNodes

```
1  // Add any node that couldn't be added in updateIndexElement.
   private void addOrphanNodes(MultiMap<String, Element> newNodeMap) {
3      Iterator<String> xPaths = newNodeMap.keySet().iterator();
       while (xPaths.hasNext()) {
5          String xPath = xPaths.next();
           IndexElement parent = findIndexElement(xPath);
7          if (parent == null) {
               throw new RuntimeException("Unable to find node " + xPath
9                      + " to add child nodes to.");
           }
11         Set<Element> newNodes = newNodeMap.get(xPath);
           xPaths.remove();
13         for (Element e : newNodes) {
               parent.normalizeElement(e);
15         }
       }
17     if (!newNodeMap.isEmpty()) {
           throw new RuntimeException("Some new child nodes were not added.");
19     }
   }
```

Figure 3.8: Example File 3

```
<foo>
  <bar/>
</foo>
```

To rectify this problem, this method does a post update check that all new child nodes were actually added, and adds them if necessary.

The method cleanDocument(fig. 3.10) was also added to resolve a problem with the simpler algorithm used initially. Namely that on occasion it would create nodes with the same VSTART/VEND timestamps. That is to say, nodes which were created and deleted in a single update. This bug was caused by a node simultaneously receiving new child nodes, and after that receiving a update which necessitates creating a new tagged node, like changing the tag name. When this happens, a node could get the same VSTART/VEND date, which from a logical standpoint means that it never really existed except as a temporary value in the v-file generator. In order to remove them, a simple recursive update of the tree checks VSTART/VEND and if they are equal, permanently deletes the node from the Index.

The method assertEquals(fig. 3.11) is a post-update sanity check. After all other updates are applied, but before update returns, this method will check if the update was performed correctly. Specifically, it will assert that the root element of the latest version of the document and that of the Index are deeply value equal, if one ignores non-live elements. This ensures that the Index algorithm works correctly and that a malformed v-file is never saved to disk.

Figure 3.9: Example File 4

```
<foo>
  <baz/>
</foo>
```

Figure 3.10: Index.cleanDocument and cleanNode

```
private void cleanDocument() {
    for (IndexElement child : this.getRootElement().elements(false)) {
        cleanNode(this.getRootElement(), child);
    }
}

/* Removes nodes with the same VSTART/VEND time,
 * and sort deleted nodes to the bottom of the file*/
private void cleanNode(IndexElement parent, IndexElement child) {
    if (child.created() == child.deleted()) {
        parent.eraseChild(child);
    } else {
        if (!child.isLive()) {
            parent.eraseChild(child);
            parent.appendChild(child);
        }
        for (IndexElement childOfChild : child.elements(false)) {
            cleanNode(child, childOfChild);
        }
    }
}
```

Figure 3.11: Index.assertEquals

```
private void assertEquals() throws IOException {
    if (!this.getRootElement().isEqualElement(
            currentVersion.getDocumentElement())) {
        this.writeToFile(new File("./err.xml"));
        throw new RuntimeException(
                "Index does not match latest version of file."
                + "\nFaulty index dumped to err.xml.");
    }
}
```

4

Results

When the v-file generator was tested, open-source projects on SourceForge that happened to contain XML files were used as input. This was done to ensure access to realistic data to provide as input to the generator. In this section, I will demonstrate two such test runs.

These test runs are merely an informal demonstration that the program works and scales acceptably to large input sizes. They should not be construed as a benchmark or proof-of-correctness.

In the first test run, the compiled executable is started with the parameters given in fig. 4.1, and it outputs what is given in fig. 4.2. If one now opens XML-indices/prerelease_maps.xml, the text listed in figure 4.3 file will be found.

Figure 4.1: Test 1 Parameters

```
svn://svn.code.sf.net/p/tripleamaps/code/trunk xml-indices prerelease_maps.xml
```

Figure 4.2: Test 1 Output

```
Indexing prerelease_maps.xml
Fetching repository history...done.
Normalizing first version of prerelease_maps.xml...done.
Indexing version 45 of prerelease_maps.xml...done.
Indexing version 56 of prerelease_maps.xml...done.
Finished indexing prerelease_maps.xml!
```

Figure 4.3: Test 1 File

```
<games docVersion="56" vend="NOW" vstart="41">
  <game vend="NOW" vstart="41">
    <url vend="NOW" vstart="56">...</url>
    <mapName vend="NOW" vstart="41">Battle_of_Jutland</mapName>
    <description vend="NOW" vstart="45">...</description>
    <version vend="NOW" vstart="41">1.7</version>
    <url vend="56" vstart="41">...</url>
    <description vend="45" vstart="41">...</description>
  </game>
  ...
```

The content of /games/game/description and URL and /games/game after the first have been left out for brevity.

One can then query this file for the information contained within. For example, we can run the XPath in figure 4.4 to find the mapName elements which are still live as of version 56. Which will result in the output in figure 4.5. Comparing with version 56 of the file in question, we can see that this is correct.

Figure 4.4: XPath Query

```
/games/game/mapName[@vend="NOW"]
```

Figure 4.5: Query Results

```
<mapName vend="NOW" vstart="41">Battle_of_Jutland</mapName>
<mapName vend="NOW" vstart="41">World War II Pacific</mapName>
<mapName vend="NOW" vstart="45">FeudalJapan</mapName>
<mapName vend="NOW" vstart="56">D-Day2</mapName>
```

Having thus demonstrated basic functionality of the implementation, a more ambitious test is attempted: The program will create a v-file of the build file of a popular open-source project which as of June 1st 2013 has 3841 commits in its SVN repository and 214 changes to the file in question. The application is started with the parameters in figure 4.6.

The elapsed time for the program to execute this test was measured by using Netbeans' built-in timer. The computer running the test is a desktop PC with a E8400 dual-core processor and 4GB of RAM which is not performing any other significant activity during the test. The measurement was only done once.

Figure 4.6: Test 2 Parameters

```
svn://svn.code.sf.net/p/java-game-lib/code/trunk xml-indices LWJGL/build.xml
```

Generating a file from input of this size takes approximately two minutes and six seconds on this computer, and results in a v-file 1.2 MB in size. The build file being is approximately 20 KB in size in each version. We can then calculate (by multiplying with the number of versions) that this compares favorably to the cost of naively storing all changed versions(4.28 MB), and of all versions (76.82 MB!)

5

Discussion

To summarize, a temporal database for XML files stored in Subversion has been implemented. This was accomplished by creating a v-file generator which compresses the history of an XML document into a single XML document, a v-file, where every element is annotated with it's transaction time.

As previously mentioned, the scope and overall goal of this project has changed multiple times, both in response to changing customer needs as well as the acquiring of new information, like the XChronicler paper. This led to the project overshooting its time budget, and some features had to be cut.

The most significant such feature is that of XML namespacing. It was planned that the v-file generator would create its attributes in a separate namespace. This is not incredibly significant, as the v-file can in any case only contain those three attributes that belong to the v-file - collisions are impossible.

What is a bit more concerning though is that the namespace of the file the v-file is based on is not preserved. It is not outside the realms of probability that this can cause bugs in files that use more than one namespace with colliding element names. This is not however a weakness in the overall approach, the XChronicler algorithm can preserve namespaces, and in fact the authors do so. The only weakness is in this implementation thereof.

Another aspect of the algorithm which was not implemented is the XChronicler algorithm's optimization of mixed-text elements, e.g. XML elements with both sub-elements and text content. This will mean that edits into such a mixed-text node's text value will still need to copy the entire node, and be rather expensive as a result. But this also could be implemented given more time.

As an aside, Repos have ordered an extension of the existing software from the author. The extension will integrate the thesis software with their CMS, as well as add the missing namespace support. The application will be restructured so that the v-file generation is run as soon as a commit is pushed to the CMS, as opposed to the current

model where the v-file is only generated on request by the user. The format of the timestamps will also be generalized from the current format (which requires them to be integer versions) to general strings. This will enable the v-file generator to support many back-ends beyond just SVN.

6

Conclusion

The project goal was to enable Repos AB to search through their version controlled XML files. This can now safely be said to have been archived. Furthermore, since the v-files are themselves XML files, Repos can leverage all the tooling support afforded to XML to work with this solution. At just about 1200 lines of code, the code base of the solution implemented is minimal and eminently maintainable, and yet solved the problem it set out to solve.

Bibliography

[1] XQuery 1.0: An XML Query Language, Tech. Rep. 2, World Wide Web Consortium (Dec. 2010).
URL http://www.w3.org/TR/xquery/

[2] F. Wang, C. Zaniolo, Temporal queries and version management in XML-based document archives, Data & Knowledge Engineering (65) (2007) 234–304.

[3] R. T. Snodgrass, I. Ahn, Temporal Databases, IEEE Computer (19(9)) (1986) 35–42.

[4] Apache Lucene - Apache Solr, accessed Sep. 2013.
URL https://lucene.apache.org/solr/

[5] Apache Solr, accessed Sep. 2013.
URL https://en.wikipedia.org/wiki/Apache_Solr

[6] Apache Tomcat, accessed Sep. 2013.
URL http://tomcat.apache.org/

[7] Jetty, accessed Sep. 2013.
URL http://www.eclipse.org/jetty/

[8] Representational state transfer, accessed Sep. 2013.
URL https://en.wikipedia.org/wiki/REST

[9] Introducing JSON, accessed Sep. 2013.
URL http://json.org/

[10] Extensible Markup Language (XML) 1.0, Tech. Rep. 5, World Wide Web Consortium (Nov. 2008).
URL http://www.w3.org/TR/REC-xml/

[11] XML, accessed Sep. 2013.
URL https://en.wikipedia.org/wiki/Xml

[12] XML Path Language (XPath) Version 1.0, Tech. rep., World Wide Web Consortium (Nov. 1999).
URL `http://www.w3.org/TR/xpath/`

[13] eXist-db Open Source Native XML Database, accessed Sep. 2013.
URL `http://exist-db.org/exist/apps/homepage/index.html`

[14] BaseX | The XML Database, accessed Sep. 2013.
URL `http://basex.org/home/`

[15] S.-Y. Chien, V. Tsotras, C. Zaniolo, A Comparative Study of Version Management Schemes for XML Documents, Timecenter Technical Report TR-51.

[16] F. Wang, C. Zaniolo, X. Zhou, ArchIS: an XML-based approach to transaction-time temporal database systems, The VLDB Journal (17) (2008) 1445–1463.

[17] S. Rönnau, Efficient Change Management of XML Documents, Ph.D. thesis, University of Munich (Oct. 2010).

[18] SVNKit: [Sub]versioning for Java, accessed Sep. 2013.
URL `http://svnkit.com/`

[19] XMLUnit - JUnit and NUnit testing for XML, accessed Sep. 2013.
URL `http://xmlunit.sourceforge.net/`