FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Enabling Data Citation for XML Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Philipp Huber, B.Sc.
Matrikelnummer 0825436

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.univ.Prof. Dr. Andreas Rauber
Mitwirkung: Dipl.-Ing. Stefan Pröll

Wien, 01.10.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Enabling Data Citation for XML Data

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Philipp Huber, B.Sc.
Registration Number 0825436

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.univ.Prof. Dr. Andreas Rauber
Assistance: Dipl.-Ing. Stefan Pröll

Vienna, 01.10.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Philipp Huber, B.Sc.
Vorgartenstraße 169/3/28, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

In this thesis we address the challenge of making subsets of XML datasets stored in native XML databases citable. Within the field of scientific research experiments are often based on data that are subsets of data sources stored in databases. References to these data are problematic as databases evolve over time by being updated. This means that identified subsets do not stay the same and hence further experiments based on already carried out experiments become impossible.

Using results of scientific research in the field of data citation as starting point, we develop two approaches to overcome this problem for native XML databases. The two approaches define two different ways of versioning and thus keeping track of changes made to the dataset in order to be able to reconstruct former states of the dataset. In both solutions a parser acting as middleware between the user and the database management system is presented. This parser receives all queries entered by the user and rewrites them in order to fit the chosen approach of versioning. Finally the rewritten queries are sent to the database management system and invoke the necessary actions.

Both approaches are evaluated via testing scenarios as well as performance tests on artificially generated datasets. We present two datasets showing different hierarchical structures and compare two different database management systems. Furthermore, we include two scenarios differing in the updates performed on the databases. The first one only inserts new content and the second one mainly updates and deletes existing content besides inserting new pieces of information.

We conclude this thesis with the insight that one of the implemented approaches is practicable in terms of storage overhead and query execution times. However we also show that the average query runtime of operations that insert, update or delete content increases from about 500 ms to 700 ms after 100 000 queries.

Kurzfassung

Im Rahmen dieser Diplomarbeit wird die Problematik der Zitierbarkeit von Subsets von XML Datensets in nativen XML Datenbanken analysiert und neue Ansätze entwickelt. Wissenschaftliche Experimente und somit Publikationen, die deren Ergebnisse zusammenfassen und diskutieren, basieren oft auf Subsets von Datensets. Diese Daten zu referenzieren wird zu einer Herausforderung sobald sich die Datenbank im Laufe der Zeit ändert, da somit auch identifizierte Subsets nicht gleich bleiben. Folglich werden weitere Experimente, die auf bereits durchgeführten Experimenten aufbauen, unmöglich gemacht.

Basierend auf bereits bestehenden wissenschaftlichen Arbeiten auf dem Gebiet der Data Citation entwickeln wir zwei Ansätze, die eine Lösung für das beschriebene Problem im Kontext von nativen XML Datenbanken darstellen. Die beiden Ansätze definieren zwei verschiedene Wege Datensets zu versionieren und somit jegliche Änderungen zu erfassen, um frühere Versionen des Datensets wiederherstellen zu können. Wir präsentieren einen Parser als Schnittstelle zwischen BenutzerIn und Datenbank Management System. Dieser Parser empfängt alle Queries, die von BenutzerInnen abgeschickt werden und generiert dem gewählten Archivierungsansatz entsprechende Queries. Abschließend werden diese neu generierten Queries an das Datenbank Management System geschickt um sowohl das gewünschte Update als auch die Historisierung der Datenbank durchzuführen.

Die Evaluierung beider präsentierter Ansätze wird mit Hilfe ausgewählter Testszenarios und Performancemessungen anhand zweier künstlich generierter Datensets, die sich durch ihre hierarchische Struktur unterscheiden, durchgeführt. Des Weiteren werden zwei Datenbank Management Systeme vorgestellt und verglichen und schließlich zwei Szenarios ausgewählt und durchlaufen um Unterschiede aufzuzeigen. Das erste Szenario wird lediglich neue Daten hinzufügen, wohingegen der Fokus des zweiten Szenarios auf dem Editieren bzw. dem Löschen bestehender Datensätze liegt.

Im Zuge der Analyse der Performancemessungen erweist sich einer der beiden präsentierten Ansätze als praxistauglich, da sowohl der zusätzliche Speicheraufwand als auch die Laufzeiten der Queries in einem akzeptablen Rahmen bleiben. Es muss jedoch auch erwähnt werden, dass sich die durchschnittliche Laufzeit der Queries nach 100 000 Operationen, die neue Daten hinzufügen bzw. alte Daten editieren/löschen, von run 500 Millisekunden auf rund 700 Millisekunden erhöht.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The authors of [59] identified that data experiments are based on are often just seen as an adjunct to publications. Publishing these data and thus making not only outcomes of experiments but also their input publicly available makes secondary analyses possible. This means that other researchers are able to re-execute the experiments and receive the same output [60]. In order to enable this, the problem of data being seen as adjunct has to be overcome.

Besides the possibility of re-executing experiments it is also possible to execute new ones based on data and observations that have already been used by other researchers. Two presentation slides of the author of [67], which are depicted in Figure 1.1, provide a good overview over today's scientific data lifecycle and what it should or may look like one day. The two images show that nowadays data is collected with instruments and acquisition tools. Afterwards they are stored and analysed. The output is a scientific article or paper and underlying data is discarded due to the fact that the storage is either only temporal (in case of data not being made available) or the data source changes over time and former states cannot be derived any more. A good standard to aim for would be the scientific data lifecycle depicted on the right side of the figure. In this case certain data about experiments are stored in persistent repositories. Studies using these data can reference them in a persistent way.

In this thesis we are trying to make experiments reproducible and make subsets of XML datasets citable. This is done with the help of native XML databases. We present solutions for databases as database management systems offer all the functionalities for editing XML files and datasets out of the box.

A special challenge when dealing with XML data instead of data organised in relational database management systems (RDBMS) is the hierarchy that has to be taken into consideration. If a piece of information is deleted for example, all the children belonging to the information

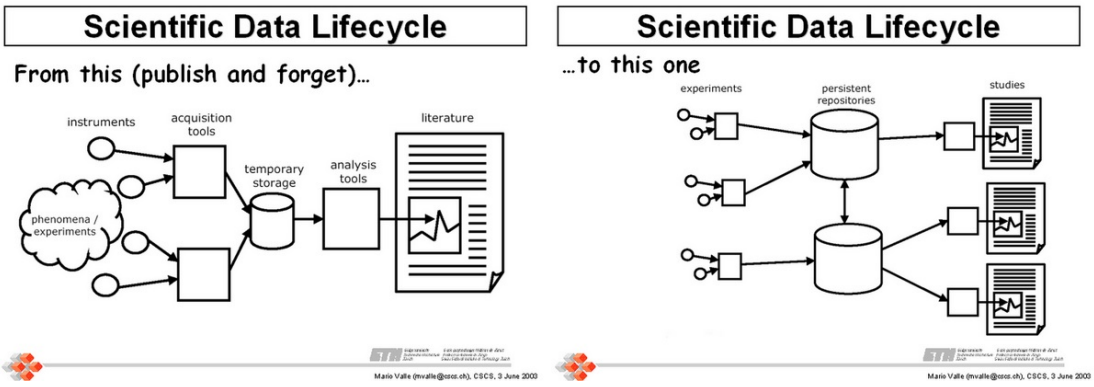

Figure 1.1: Scientific Data Lifecycle - how it Should Evolve [67]

that has to be deleted have to be deleted as well. This makes dealing with that sort of data more difficult.

1.2 Definition of the Scientific Problem

Although data sharing is not new, it has not been common practice for researchers to publish data or subsets of data that they used in their experiments for a long time [61]. In this upcoming era of science it becomes more likely to publish research data for the purpose of documentation and archival storage. Thus researchers can build their future work not only upon existing publications but also benefit from data sharing. Without publishing and making data scientific research is based on accessible, the data often get lost because outcomes are only released in the form of paper based publications [33]. Thus making the data accessible is a challenge that has to be overcome. Rerunning experiments becomes possible when not only the output but also the data experiments are based on are published. If this is achieved constructive research is enabled [54].

If the subsets of data sources the experiments are based on are archived and offered as a download, it is sufficient to reference the data. Hence it is possible to re-execute former experiments at a later point in time based on the archived subsets. Such file based archiving technologies are simple to realize but occupy a lot of disc space. It is a better practice to reference the used data sources and to provide information on how the subsets were derived. In the context of databases information about the database itself and the queries that were used to identify the subsets have to be provided. A challenge arises when databases change over time and hence also the identified subsets do not stay the same. Queries deriving subsets from databases in the context of experiments and scientific work can be archived but when certain pieces of information are edited or new ones are added, the results of the archived queries might not be the same any more when they are re-executed. Thus it has to be ensured that every single modification has to be tracked in order to be able to reproduce the state of the database at a given point in time. This means that any piece of information must not be deleted or overwritten but only disabled in the

course of performed updates. Queries, which are executed at a later point in time and address the latest/current state of the dataset do not take these disabled pieces of information into account but queries that are re-executed can still access them.

Due to the rapidly growing amount of research databases, the appropriate and accurate citation of datasets becomes increasingly important [25]. Referencing these scientific data and enabling scientists to re-execute experiments and build up new research based on the results of other publications is important. Anyways, the way to consistent storage and citation of data requires a cooperation among funding agencies, scientists and engineers.

To overcome and to present a solution to the challenges stated above, the research objective of this thesis was to find a way of making subsets of XML data sources stored in native XML database management systems (DBMS) citable. Concluding the research questions can be formulated as follows:

How can XML data sources used in native XML databases be made identifiable to foster re-execution of queries and to document the state of the database at given points in time in order to make subsets of data reproducible?

1.3 Enabling Reproducible Research

The term "reproducible research" has been coined by Jon Claerbout, a professor at Stanford University in 2000 [26]. This wording is used when it comes to the idea of "replication by other scientists" in reference to computations done in scientific research. The basic principle of reproducible research is that independent researchers have to be able to re-execute a certain experiment under the same conditions and receive the same output [60]. Thus the ultimate product is not only the published paper itself, but also the whole environment that leads to the results presented in this specific paper (e.g. the dataset, the software, etc.) [40].

Sharing data scientific work is based on has the advantage of making secondary analyses possible and facilitating them [72]. If underlying data is not published, the progress in computational science is often hindered by the fact that published results based on the data cannot be reproduced or verified [73]. With verification being made possible, errors in analyses can be identified, reported and corrected. Furthermore this process may even help preventing such errors in the future [72].

When experiments are run, subsets of data sources are selected, created and used as input. The results of these experiments are either new data or updates to the underlying data source. If these results are not published and made publicly available the scientific progress is hindered and new experiments cannot be performed on the achieved results.

With publishing data experiments are based on, not only outcomes of research but also the steps that were made in order to achieve them are made available to a broad audience [23]. Thus

transparency and participation in the scientific process is made possible. When data is shared and users are able to reuse them, the data can be accessed at any time, by anyone and can be used for any purpose. It is thus possible to identify, retrieve, replicate and verify underlying data [47]. This means that it is ensured that outcomes of experiments are comparable because they can be re-executed at any point in time.

To ensure experiments can be re-executed not only hardware to store the data but also software that helps users to actually re-execute the experiments is needed [49]. If such a software is not provided users at least have to be provided descriptions on how to retrieve the needed subsets and how to re-execute experiments.

The authors of [21] identified the following six steps that have to be taken in order to enable data sharing.

1. Collecting examples of successful data sharing
2. Drafting of a model
3. Testing of the model for completeness
4. Updating of the model
5. Identifying of the most important barriers and drivers
6. Deriving recommendations

The Yale Law School Roundtable on Data and Code Sharing identified six steps that scientists could follow to generate reproducible results [73].

1. Source-code and data used for the experiments are archived
2. Each version of released code gets a unique ID assigned (version control system)
3. A statement describing the computing environment (software, ...) is given
4. Open licensing for code is chosen
5. An open access contract for published papers is used
6. Data and code are published in non-proprietary formats

1.4 Scientific Methods

This section is divided into two subsections. The first one, Section 1.4.1, will provide an insight into the methodology used to achieve the expected results and the second part, Section 1.4.2, will cover the evaluation criteria of the goals and how they are checked.

1.4.1 Methodology

The six subsequently listed steps represent the methodological approach and thus the procedure that will be followed while writing this thesis. After going through these steps the objectives of this thesis should be reached.

1. **Literature review:** In the first part, literature will be looked up and reviewed to provide an understanding of topics that are connected to XML databases, data citation and reproducible research.

2. **Selection of XML DBMS:** As a second step two database management systems will be chosen. These two systems will be used in the empirical part of this thesis.

3. **Selection of XML data:** As a third step proper XML-datasets will been chosen. The datasets will be presented and described in detail later on.

4. **Identification of approaches to data citation for XML databases:** The fourth part will consist of conceptual thinking about the topic of data citation for XML databases. Thus models including features like versioning and timestamps will be developed. Possible approaches will be listed, described and demonstrated with the help of two chosen database management systems.

5. **Implementation of own solutions:** After approaches have been identified, they will be implemented and evaluated.

6. **Analysis of the identified approaches:** In the last part the identified approaches will be analysed and compared. Advantages and disadvantages will be listed and checked against each other.

1.4.2 Evaluation

After finishing the scientific work we will evaluate our results in terms of performance and behaviour. Additionally we will check whether they meet all the requirements that we will identify in the listing below. Talking about the performance we will find that one of our two approaches taking the job of the versioning of datasets is practicable regarding performance and disc space needed and the other one is not.

1. Every experiment users want to archive has to be reproducible. This means that every query identifying subsets of databases have to be reproducible at any point in time. With this requirement it is possible to retrieve the same subset as originally returned at a later point in time.

2. Newly entered data have to be valid and meet the specifications of the respective approach.

3. All the changes of datasets have to be tracked and versioned. This means that a kind of versioning has to ensure that no pieces of information are lost in order to be able to reconstruct subsets created at earlier stages of the dataset.

After the definition of the requirements the results have to meet, it is important to determine the steps of evaluation. The fact that every experiment has to be reproducible and archived queries have to return the same subset at any point in time not minding the changes that happened to the dataset will be evaluated with the help of test scenarios. The second fact, evaluating whether newly entered data is valid and according to the respective approach, will be done by editing certain data records in the databases. Hence the evaluation of this requirement is built on controlled experiments. The testing of appropriate error-handling will be done with the help of test cases as well.

1.5 Thesis Outline

This thesis is organized as follows. After this introduction, Chapter 2 ("Related work") will contain related work and will define basic terms that we will use throughout this thesis. Additionally we will summaries the scientific work that has been done in this field of study until present. The next two chapters, Chapter 3 ("XML & Description of Used Datasets") and Chapter 4 ("Databases for XML"), will cover the topics of XML including the description of the datasets we will use in this thesis and the topic of databases for XML. The latter of these chapters will provide an insight into the topic of database management systems, describe the two DBMS we will use in this thesis, namely "BaseX" and "eXist-db", and compare them in terms of behaviour when it comes to queries that update the database. After all the theoretical concepts and other scientific works being described, Chapter 5 ("Devising a Versioning and Subsetting Solution") will contain the empirical part of this thesis. This chapter will cover exact specifications and descriptions of the concepts that will be used in order to run the two approaches presented by us, the description of the approaches themselves as well as specifications of the prerequisites and assumptions. Chapter 6 ("Evaluation of the Versioning and Subsetting Solution") will cover the topic of performance testing of the approaches implemented by us. Concluding this thesis, Chapter 7 ("Summary and Future Work") will provide a brief summary of the work that has been done in this thesis and provide an outlook for possible future work.

All the documents that were created during the writing of this thesis including links to the Java projects can be accessed and downloaded from the homepage `http://www.ifs.tuwien.ac.at/dp/datacitation_xml`.

CHAPTER 2

Related work

This chapter will provide an insight into the topics of data citation and persistent identifiers. In Section 2.1 we will describe the principles of data citation and what is needed in order to make subsets of datasets citable. In Section 2.2 we will present already existing approaches to the topic of versioning for XML documents. Section 2.3 will explain persistent identifiers and will additionally list several concepts of persistent identifiers. Concluding this chapter, Section 2.4 will provide a brief summary and will highlight the importance of the presented topics.

2.1 Data Citation

In the context of scientific research and papers being published one has to cite used references appropriately and accurately in order to show which sources of information have been used [2]. A proper citation permits readers to check the underlying literature and allows them to follow up on aspects of the work. This being defined we can take a step further and provide a brief definition of the term "data citation" with the help of the following quotation.

> *"Data citation refers to the practice of providing a reference to data in the same way as researchers routinely provide a bibliographic reference to printed resources [7]."*

Citing data or subsets of data is a step in the right direction but it is not enough to just refer to them as their location of storage might change over time [34]. URL (Uniform Resource Locator) hyperlinks are used to identify and link content stored on the internet. The problem is that a URL provides specific location details for a document but not the document itself. Given the fact that a resource can be relocated and consequently the storage location of the document can change, the denoted URL does not link to the right resource any more. Thus the original resource becomes inaccessible to the user. Gradually more and more hyperlinks become 'broken' links. Therefore improved citation practices are required to minimize future loss [43]. Users

should be able to access resources not minding whether they have been relocated or not.

To overcome this problem of resources getting relocated, several approaches to the concept of persistent identification were elaborated and implemented. An insight into the topic of such persistent identifiers including a brief overview over different types is given in Section 2.3.

Talking about citing subsets of datasets one might think of archiving data with the help of persistent identifiers [54] [55]. Assigning PIDs to portions of data - e.g. database cells when it comes to SQL databases - and not directly editing but storing multiple versions of data would be sufficient. With this approach two big problems arise - first of all the huge number of PIDs that would be needed and secondly the disc space comes into play. Considering the fact that datasets grow and are updated in the course of time, assigning PIDs to static data dumps is not adequate. Thus data citation within the scope of databases requires versioning and timestamping.

2.1.1 An Introduction to Data Citation

Several organizations and institutions wanted to achieve the goal of citing data properly and thus built up standards for doing so [66]. The provided pieces of information like the author, the year or a title are similar to the ones being provided when publications and other scientific work are cited. Due to the fact that multiple standards for citing data exist, problems like the unambiguous identification of such data appear [3]. Techniques vary from field to field, archive to archive and even from article to article.

In order to make citation of datasets a well known standard, a lot of effort has to be made by national as well as international agencies [50]. The Organisation for Economic Co-operation and Development (OECD) defines two steps to make this possible.

1. A policy for data citation has to be formulated and published.

2. A culture of data citation inside and outside of organisations has to be encouraged.

Compared to citing papers or journal articles, citing data or subsets of data is more challenging because they can change over time and thus are more dynamic in terms of content and version [66]. The fact that researchers might modify or add information to the datasets makes them evolve over time. The authors of [54] identify the following four requirements to unambiguously cite subsets of a database that changes over time.

1. Subsets of data collections have to be referencable

2. Handling of dynamic data has to be possible

3. Enabled scalability

4. Transparent implementation

The first point addresses the reuse of data [54]. On the one hand it has to be possible to re-execute experiments and on the other hand performing new analyses of experiments and thus making new ones based on old subsets of data need to be enabled. The second requirement is about the handling of dynamic data because of the fact that databases can change and evolve over time. The third requirement demands scalable methodologies which can handle all the different sizes of databases. The last point claims usability as a solution is only going to be accepted if the solution is transparent and reasonable.

The authors of [39] already stated in 1986 that it is not efficient to model only the present state of the world. Hence the concept of versioning environments was developed. With the help of primitive mechanisms like timestamps, which are capable of storing system times implying when updates were performed, complex version structures can be built in such version environments.

Given the problems discussed previously and building on the above stated requirements it is the goal of this thesis to find a way of making subsets of XML databases citable. Following the recommendations of the authors of [57] we face two main challenges. First of all the dataset has to be versioned with the help of timestamping in order to be able to keep track of changes happening to the dataset. Secondly a mechanism that allows to archive queries that derive subsets and makes their re-execution possible has to be provided.

Concerning the versioning and timestamping the term of temporal databases, which has been around for a long time [62] and is still present in current research [44], is to be explained. Temporal databases can be defined as databases that maintain past, present and future states of data [4]. The different states are maintained with the help of timestamps showing when certain pieces of information were valid or stored in the database. In temporal databases two concepts, namely valid time and transaction time, are differentiated [64] [41]. Valid time represents the period of time during which a fact is considered to be true with respect to the real world. This means that the valid time can also be in the future if a fact is considered to be true in the future [63]. When talking about transaction time the period during which a fact is stored in the database is meant. An example for this is a database holding contracts - the time a contract is valid would in this example be the valid time and the time the contract is stored in the database would be the transaction time.

The following section will provide a description of a model developed for dynamic data citation in relational databases.

Model for Dynamic Data Citation in Relational Databases

The authors of [54] present an approach for the citation of dynamic data in the context of relational databases. They identified that all the UPDATE, INSERT and DELETE statements have to be tracked to be able to trace all the changes made to the dataset. Thus UPDATE and DELETE operations must not update or delete the respective data but only mark the records indicating that they have been updated or deleted. With the help of timestamps using the concept of transaction

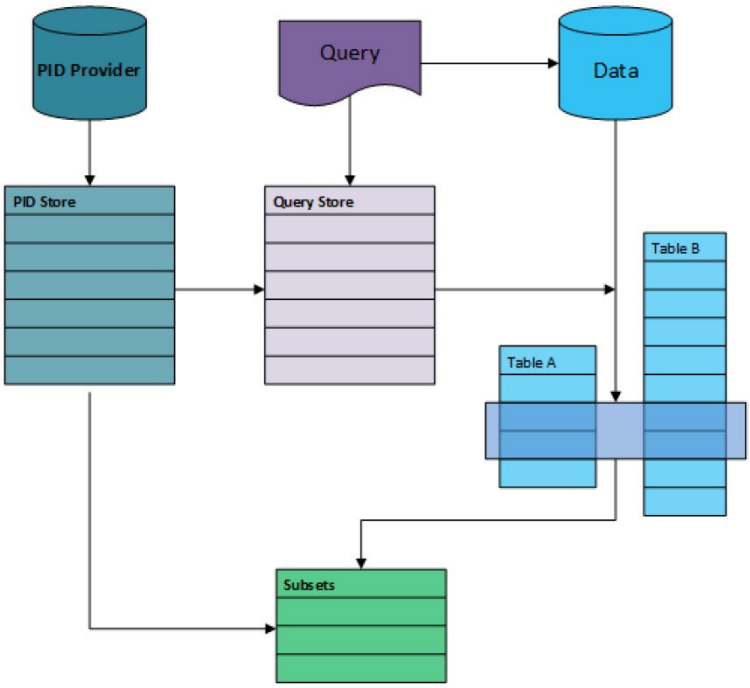

Figure 2.1: Approach presented by the authors of [54]

time it can be derived when data records were created and when they stopped being valid. Hence previous states of data can be derived.

Due to the fact that the creation of timestamps and the versioning / archiving is done automatically, the user does not have to invoke any extra actions. When a subset of data is queried for an experiment, the data as well as a PID is returned. With this approach it is possible to re-execute queries and to receive exactly the same subset that was identified when the query was originally run regardless of any changes that happened to the dataset.

Figure 2.1 depicts the concept of the component allowing the archiving of subset identifying queries and their re-execution - the query store ("Query Store"). It stores archived SELECT queries together with a timestamp representing their time of execution and a PID that is received from the "PID Provider". With the help of this PID archived queries can be identified and the subsets defined by them can be derived by using the version of data that was valid at the time of the original execution of the query.

2.2 Versioning Approaches for XML

In this section we will present four approaches to data citation in the context of XML. Thus scientific research in this area that has been done until present, will be highlighted. For further

readings please check the respective references. At first, an approach to represent time in XML will be described in Section 2.2.1. After that, Section 2.2.2 will describe the approach of schemes for multiversion XML documents and Section 2.2.3 will cover an approach to help detecting changes in XML documents. The last approach that will be be presented in this section is the one of XArch in Section 2.2.4.

2.2.1 Presenting Time in XML

One approach for representing time in temporal XML databases is proposed by the authors of [4]. Leaf data nodes can have alternative and thus multiple values that are valid for certain time periods. The concept of valid times is used implying what value was valid at what point in time. This means that all the former states of these leaf nodes are tracked. A non leaf element node is defined as a container holding further child-element, attribute and text nodes. The valid time of such non leaf element nodes can be derived with the help of all the children's valid time.

For this approach two versions, namely "full implementation" and "simplified implementation" exist. Both of these two versions will be described with the help of an example. The considered XML file will be described in Figure 3.1 on page 24, but will be depicted in Listing 2.1. The two Listings 2.2 and 2.3 depict the XML file being versioned with the respective version of this approach and after two performed updates. The first update replaces the text value of the "segLabel" element node with "KIJOGH" and the second one inserts the attribute "information" into the element node "endIndex".

Listing 2.1: Presenting Time in XML - Example XML File

```
<tmSegments>
  <tmSegment>
    <segLabel>A</segLabel>
    <beginIndex>55</beginIndex>
    <endIndex>86</endIndex>
  </tmSegment>
</tmSegments>
```

Listing 2.2 depicts the document when using the "full implementation" version. For this version three important things have to be mentioned:

1. Each element node has got an attribute named "valid", which holds the information when the element started being valid and when it stopped being valid. If the last value is set to "now" the element node is still valid.

2. All the attribute nodes are mapped to an "attribute" element node. The "attribute" element is inserted as a child of the element node that originally held the attribute node. The attribute "name" inside the "attribute" element shows the name of the attribute.

3. All the text nodes are mapped to a "stringvalue" element node.

Listing 2.2: Presenting Time in XML - Example XML File - Full Implementation

```
<tmSegments time:valid="2000-01-01, now">
  <tmSegment time:valid="2000-01-01, now">
    <segLabel time:valid="2000-01-01, now">
      <time:stringvalue time:valid="2000-01-01, 2012-05-05">A</
         time:stringvalue>
      <time:stringvalue time:valid="2012-05-06, now">KIJOGH</time
         :stringvalue>
    </segLabel>
    <beginIndex time:valid="2000-01-01, now">
      <stringvalue time:valid="2000-01-01, now">55</time:
         stringvalue>
    </beginIndex>
    <endIndex time:valid="2000-01-01, now">
      <time:attribute name="information" time:valid="2012-05-06,
         now">A2BJK</time:attribute>
      <time:stringvalue time:valid="2000-01-01, now">86</time:
         stringvalue>
    </endIndex>
  </tmSegment>
</tmSegments>
```

The example of the "full implementation" version shows that the structure of the document completely changes. Current text values and attributes cannot be queried normally as they are stored in newly generated "attribute" and "stringvalue" elements. The "simplified implementation" version is considered as an optimized version of this implementation. Listing 2.3 depicts the document when using the "simplified implementation" version. With this version the original form of the document can be retained. It can be seen that this version only uses the "valid" attribute in order to show the version history and the validity of elements. Nodes that are updated are duplicated and exist in two versions from this point on.

Listing 2.3: Presenting Time in XML - Example XML File - Simplified Implementation

```
<tmSegments time:valid="2000-01-01, now">
  <tmSegment time:valid="2000-01-01, now">
    <segLabel time:valid="2000-01-01, 2012-05-05">A</segLabel>
    <segLabel time:valid="2012-05-06, now">KIJOGH</segLabel>
    <beginIndex time:valid="2000-01-01, now">55</beginIndex>
    <endIndex time:valid="2000-01-01, 2012-05-05">86</endIndex>
    <endIndex information="A2BJK" time:valid="2012-05-06, now">
       86</endIndex>
  </tmSegment>
</tmSegments>
```

This versioning approach was essential as input for this thesis. Approach I - “Branch Copy” is similar to the “simplified implementation” with the difference that we use two different attributes that imply the start and end time of their validity. Approach II - “Parent Child” was inspired by the “full implementation” as it tries to minimize the redundancy of information. However, when using our approach text and attribute nodes do not have to be represented as element nodes.

The authors of [74] present another approach to representing time in XML documents. In this approach valid times are represented with the help of “validTime” element nodes, which are held by each element in the dataset. Listing 2.4 depicts an example of such a “validTime” element. This example shows that the respective information started being valid on “20.05.2010” and is valid until present (implied by the term “now”). Our second approach we present in this thesis, Approach II - “Parent Child”, uses versioning block being similar to this concept with the difference that that they only serve the purpose of versioning text and attribute nodes. Furthermore the versioning blocks presented by us directly store the specific values for each point in time. This makes our approach perform better in terms of required storage space.

Listing 2.4: Adding Valid Time to XPath - “validTime” Element

```
<validTime>
  <time>
    <begin>
      <day>20</day>
      <month>05</month>
      <year>2010</year>
    </begin>
    <end>
      now
    </end>
  </time>
</validTime>
```

2.2.2 Schemes for Multiversion XML documents

The approach presented by the authors of [70] deals with successive versions of XML documents with the help of a concept called V-Document. A V-Document uses the concept of valid times (already described in Section 2.2.1) adding the two attributes “vstart” and “vend” to each element node. These two values represent the valid version interval of the respective element. The attribute “vstart” holds the date when the element was added to the document or database and “vend” represents the date when the element stops being valid. To ensure that an element is valid in every new version, its value can be set to “now”.

To make the usage of this concept clearer, an example will be given subsequently. The considered XML file will be depicted in Listing 2.5.

Listing 2.5: Detecting Changes in XML Documents - XML File

```
<tmSegments>
  <tmSegment>
    <segLabel>A</segLabel>
    <beginIndex>55</beginIndex>
    <endIndex>86</endIndex>
  </tmSegment>
</tmSegments>
```

The according V-Document to the document stated in Listing 2.5 is given in Listing 2.6. The example does not show the usage of attributes, but they can be handled easily via subnodes and the tag “isAttr”.

Listing 2.6: Detecting Changes in XML Documents - XML File

```
<tmSegments vstart="2012-05-05" vend="now">
  <tmSegment vstart="2012-05-05" vend="now">
    <segLabel vstart="2012-05-05" vend="now">A</segLabel>
    <beginIndex vstart="2012-05-05" vend="now">55</beginIndex>
    <endIndex vstart="2012-05-05" vend="now">86</endIndex>
  </tmSegment>
</tmSegments>
```

Three types of operations manipulating and updating data are identified, namely “update”, “insert” and “delete”. How these three operations are handled will be explained in the listing below.

- **Update:** When an update is performed, the element node is copied and the new one is updated accordingly. The attribute “vstart” of the new node is set to the current version (represented by a timestamp) and “vend” is set to “now”. Additionally the attribute “vend” of the old node is set to the last version (again represented by a timestamp) before it was changed.
- **Insert:** When an element node is inserted, the “vstart” attribute is set to the current version number and “vend” is set to “now” to imply that it is still valid.
- **Delete:** When an element node is deleted, the attribute “vend” is simply set to the last version the element was valid in.

The authors of [70] list the following advantages of this method.

1. No storage redundancy because nodes staying the same are represented by the valid version intervals.
2. Temporal queries can be easily formulated and executed with the help of standard XML query languages.

Talking about the first advantage of the above stated list it has to be said that redundancy is not completely avoided. As it has already been described, element nodes are copied when they are updated. In this case all the children of the element node are copied as well which means that each of the children exists two times. The old children are disabled and thus inaccessible but they still exist in the database increasing its size. This redundant storage of information should be optimised in order to decrease the required storage space.

Comparing this approach to the "simplified implementation" of the approach presented in Section 2.2.1 shows that they are very similar - the big difference is that the validity of element nodes is shown with the help of two separate attributes instead of a single one. Our first approach named "Branch Copy" implements this concept of versioning.

2.2.3 Detecting Changes in XML Documents (XML Diff)

After a lot of research and work on diff algorithms for relational data, e.g. [42], or even for tree data, e.g. [71] and [18], has been done, the authors of [19] present a diff algorithm for XML data and thus an approach to detect changes in XML documents. A diff algorithm serves the purpose of calculating and displaying differences between two states of two files.

This diff algorithm tries to identify (large) subtrees that were left unchanged between the old and the new version immediately after changes happened to a certain document [19]. After the unchanged subtrees are detected, the algorithm tries to match more nodes by taking a look at ancestors and descendants of already matched nodes. After the matching is done, changes are identified and represented with the help of a delta. The used delta has previously been identified by the authors of [46] and is based on insertion and deletion of subtrees or nodes, updates of text nodes or attributes and moves of subtrees or nodes. It has to be mentioned that the delta is complex and contains redundant information as it contains both the new as well as the old value in case of an update. In practice, any version of the document can be reconstructed, given a certain version of the document and the corresponding deltas.

One big issue the algorithm is facing is the management of positions and thus the unique identification of nodes. To achieve persistent identification of nodes, every node of the first version of the XML document is assigned a unique identifier, for example its position in postfix notation. When an update is performed and nodes stay the same, they keep their unique identifier. New nodes are assigned new unique identifiers.

For better understanding of the concept of the delta, an example is provided in Figure 2.2. A smaller version of the XML file has already been used in Section 2.2.2 and is thus not listed again. Figure 2.2 depicts the tree structure of the XML file and the changes that are made to it.

Listing 2.7 shows the delta for the changes stated above is given. The position of the initial nodes is stored in the attribute "XID-map" and is derived from the position within the tree in postfix notation.

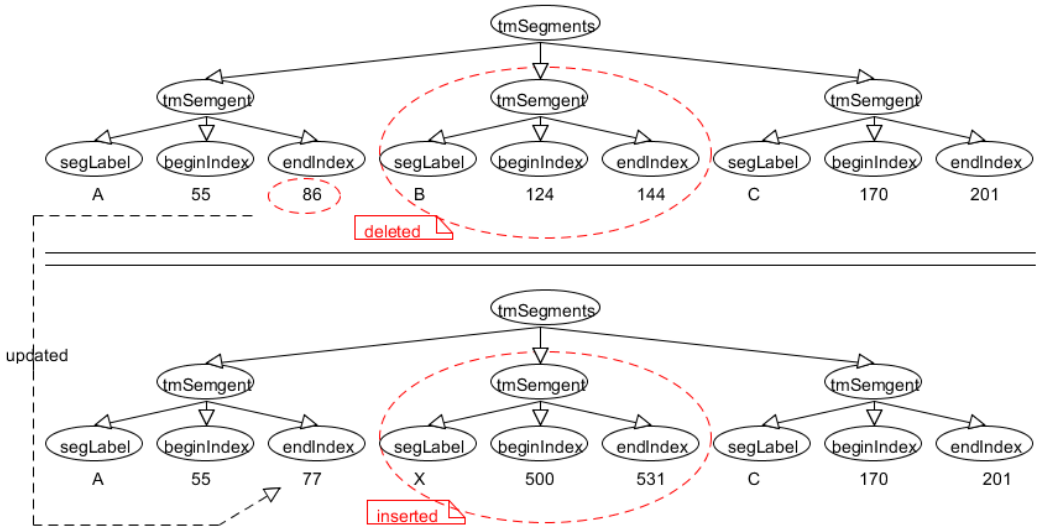

Figure 2.2: Tree Structure of the XML document and changes made to it

Listing 2.7: Detecting Changes in XML Documents - Delta Example

```
<delete XID=14 XID-map="(8-14)" parentXID=22 pos=2>
  <tmSegment>
    <segLabel>B</segLabel>
    <beginIndex>124</beginIndex>
    <endIndex>144</endIndex>
  </tmSegment>
</delete>
<insert XID=29 XID-map="(23-29)" parentXID=22 pos=2>
  <tmSegment>
    <segLabel>X</segLabel>
    <beginIndex>500</beginIndex>
    <endIndex>531</endIndex>
  </tmSegment>
</insert
<update XID=5>
  <oldval>86</oldval>
  <newval>77</newval>
</udpate>
```

This example gives an insight into the functioning of the delta. The diff algorithm itself is not described in this section. For further information the paper [19] can be followed up. Using the approach presented in this section as input we tried to implement a kind of a diff algorithm in the form of tracking tables that hold information about all the performed updates. Due to the

reasons that are described in Section 5.1 we did not implement such an approach.

2.2.4 XArch

Xarch is an archive management system, which allows creating, populating and querying of archives holding multiple versions of databases [16] [48]. With this tool users are able to create new archives, merge differing versions of data into already stored archives and query them. Queries onto these archives are executed with the help of a declarative query language. Currently XArch is able to handle and store relational databases as well as XML data. Implemented by the University of Edinburgh Database Group [1], XArch can be downloaded via SourceForge [2].

When two archives are merged, node pairs that correspond according to their defined key values are identified [15] . Afterwards these nodes are merged together and the resulting nodes are marked with timestamps of all the nodes that were merged. Finally this procedure is repeated for all the children iteratively until the leaf nodes are reached.

To visualise the working of XArch an example is given in Figure 2.3. The upper half of the graphics shows two states of a database that are merged into one archive. The bottom half of this picture shows the archive after the merging process. Listing 2.9 shows this example in XML syntax.

In the example given in Figure 2.3 it can be seen that the root node with the name “tmSegments” existed in both versions but the second “tmSegment” only exists in version 2 and not in version 1 [15]. The first “tmSegment” element node inherits the validity of its parent node because timestamps are only stored at an element if it differs from its parents one. The only node in this branch, which is versioned, is the text content of its “endIndex” child due to the fact that the value changed.

Listing 2.8: Detecting Changes in XML Documents - Delta Example

```
<T t="1-2">
  <tmSegments>
    <tmSegment>
      <segLabel>A</segLabel>
      <beginIndex>55</beginIndex>
      <endIndex>
        <T t="1">86</T>
        <T t="2">77</T>
      </endIndex>
    </tmSegment>
    <T t="2">
      <tmSegment>
```

[1] Compare http://wcms.inf.ed.ac.uk/lfcs/research/groups-and-projects/database - Last access: 07-08-2015

[2] Compare http://sourceforge.net/projects/xarch/ - Last access: 07-08-2015

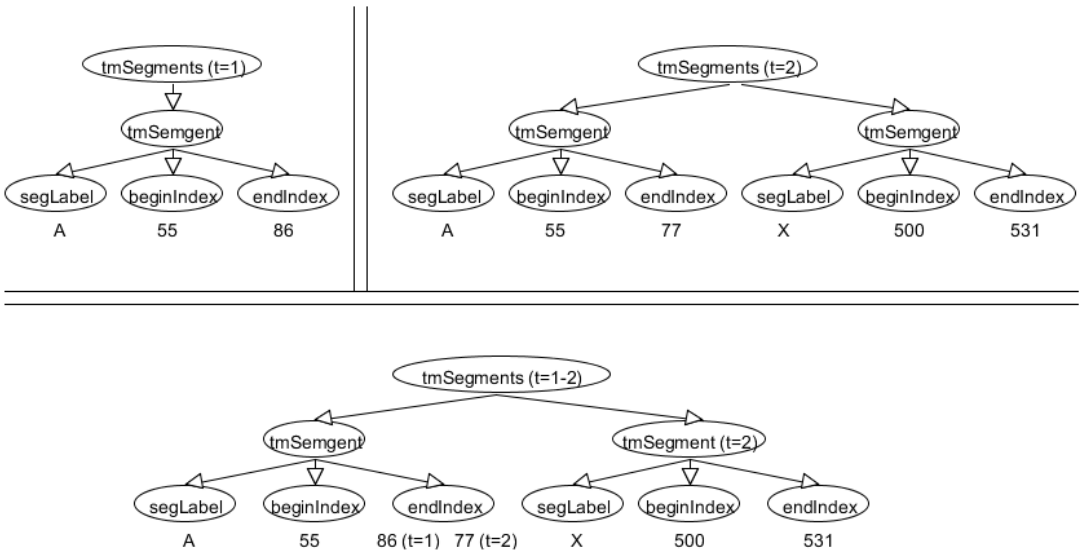

Figure 2.3: Example of a Dataset stored in XArch

```
      <segLabel>X</segLabel>
      <beginIndex>500</beginIndex>
      <endIndex>531</endIndex>
    </tmSegment>
  </T>
 </tmSegments>
</T>
```

Concluding this section, Figure 2.4 depicts the GUI of XArch. On the left side of the user interface information about archives and data sources stored within XArch is provided. On the right side in the top section queries can be defined and entered and on the bottom section the queried information is displayed.

Although we did not build our approaches based on XArch it was important to get an idea of how timestamps can be "pushed down" as far as possible in order to minimize redundancy.

2.3 Persistent Identifiers

As it has already been stated, persistent identifiers are capable of solving the problem of resources getting relocated on the web [34]. A PID identifies a resource independently from its storage location by separating the identifier from the locator and providing a mapping mechanism. Persistent identifiers are registered and serviced centrally in a database. When a specific PID is looked up, the identifier in the request is mapped and resolved so that the respective document can be shown without the user knowing the precise storage location of the document itself.

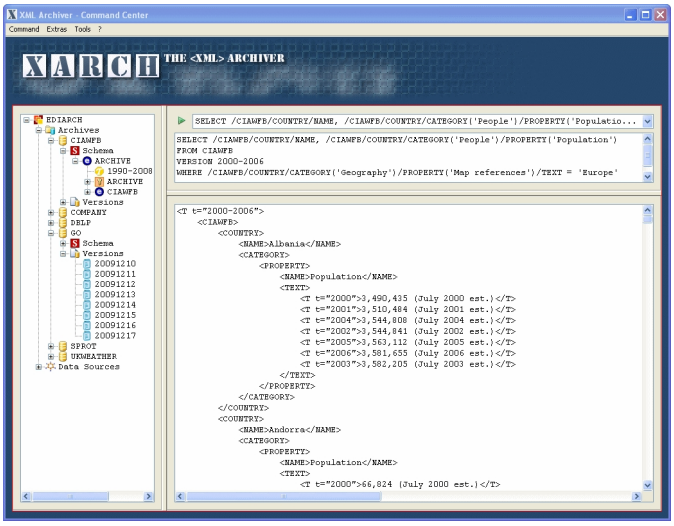

Figure 2.4: GUI of XArch [16]

"Persistent identifiers are unique identifiers for digital objects; they ensure reliable access to the relevant resources over long periods, even following system changes [27]."

Approaches to persistent identification include e.g. Digital Object Identifiers (DOI), Uniform Resource Names (URN), Persistent URLs (PURL), Archival Resource Keys (ARK), Uniform Resource Identifiers (URI) and handles [34].

In the context of persistent identifiers it is not only important to unambiguously identify a resource but it is also crucial to ensure credibility, authenticity and continuous access to it [8]. Services that are responsible for establishing the link between the documents and the identifier as well as locating and accessing the documents are called Registration Authorities (RA). The above mentioned aspects of credibility, authenticity and the fact that resources have to be accessible continuously are clearly related to the chosen RA. A very important feature covering the unambiguous identification of a resource is that a persistent identifier is globally unique forever and is never going to be re-assigned.

Consecutively four concepts of persistent identifiers will be listed and described in Sections 2.3.1 - 2.3.4. Concluding the section on PIDs, a short summary of the concepts will be given in Section 2.3.5.

2.3.1 Uniform Resource Identifier (URI)

Uniform Resource Identifiers (URI) are alphanumeric strings that serve the purpose of identifying resources on the web [45]. URIs can be divided into the two subclasses "Uniform Resource Locators" (URL) and "Uniform Resource Names" (URN).

URLs point to the physical storage location of digital objects [45]. Due to the fact that electronic resources might move without leaving a forwarding address, URLs are not adequate for persistent identification.

URNs on the other hand are based on identifying resources via names [34]. A Uniform Resource Name is structured as follows.

Listing 2.9: Detecting Changes in XML Documents - Delta Example

```
'urn:' <NID> ':' <NSS>
```

In this example the namespace identifier, NID, is a case-insensitive sequence of numbers, letters and hyphens [34]. The tag NSS stands for "namespace specific string" that uniquely identifies a resource in the specific namespace, which is defined by the NID. An example of a URN is *urn:isbn:3-938616-59-8*. In this example an international book number (ISBN) "3-938616-59-8" is mapped and represented as a URN in the "isbn" namespace.

The authors of [68] state that casting URIs into schemes like URLs or URNs is not important and can therefore be disregarded. Web-identifier schemes should generally be seen as URI schemes. Thus "http:" as well as "urn:" should be seen as URI schemes with the only difference of having different namespaces.

2.3.2 Handle System

The handle system is an implementation of the idea of URNs. It provides a method to unambiguously identify registered resources. Every Handle that is assigned is unique within the particular handle system. Each Handle consists of a prefix showing the naming authority (domain name service) and a suffix identifying individual documents. These two components are separated by a slash. The following string is an example for a handle: *15.875/file6472* where "15.875" stands for an organisation that owns the right to name documents in the handle system [45]. The second part, "file6472", identifies a specific document.

With the help of a central root server named Global handle system, the user is able to resolve all the handles independently of the domain name service where the single resources are registered [34]. Thus this central server delegates queries to all the other naming authorities.

One of the biggest implementations of the handle system is the subsequently described concept of Digital Object Identifiers [45].

2.3.3 Digital Object Identifier (DOI)

In October 1998 the concept of DOIs was introduced [34]. Since then the International DOI Foundation (IDF) has managed and controlled the database of all the Digital Object Identifiers (DOI). The general idea behind DOIs is that they provide administrative workflows and schemes for the identification and management of digital objects. Currently publishers like IEEE, MIT

Press, Scholarpedia or Springer use the concept of DOIs. [3]

As other approaches to persistent identification DOIs need a centralised registration and resolution system [45]. A service making the resolution possible has to be provided because web-browsers are not able to resolve DOIs on their own. The concept of DOIs uses the same syntax as the handle system and is fully compatible with it. When using DOIs users are given the possibility to add metadata that describe the referenced content [52].

An approach that is based on the concept of DOIs is a contract signed in 2009 to ensure cooperation between parties like the British Library, the Library of the ETH Zurich and the German National Library of Science and Technology [14]. This contract helps to establish a not-for-profit agency giving organisations the opportunity of registering datasets and assigning persistent identifiers to them. Thus these datasets are citable and can be treated as independent scientific objects. In this approach a Global DOI registration agency (RA) named "DataCite" was established and has since been supported by libraries instead of publishers. The project puts forward especially the advantage of reducing infrastructure costs and the integration of existing infrastructures.

2.3.4 Persistent URL (PURL)

One of the first implemented approaches for persistent identifiers is the concept of persistent URLs (PURL) [34]. They were implemented in 1996 in order to advance cataloguing practices for resources stored on the internet. A PURL is a URL differing in the fact that it does not directly point to an internet resource but resolves the address to the actual URL that points to the resource. The access to the requested file is established directly by the client.

This means that referenced resources and thus underlying web addresses of these resources can change over time without them becoming inaccessible [35]. A PURL is matched to the actual URL by the PURL resolution service and returns the URL to the client as a standard HTTP (Hypertext Transfer Protocol) redirect. Hence the concept of PURLs ensures continuity of references to resources stored on the internet not minding whether they may migrate from machine to machine or not.

2.3.5 Conclusion

Concerning the area of persistent identifiers different technological solutions - four of them were listed in the sections above - have been developed. Research is still trying to find a best case solution to the problem of citation and persistent identifiers. The authors of [9] for example highlight the problem of several persistent identifiers of different PID systems getting assigned to one single resource and propose the implementation of the Interoperability Framework (IF) for persistent identifiers (PIDs). The main goal of this framework is the creation of a cross-

[3] Compare `http://www.crossref.org/01company/06publishers.html` - Last access: 05-12-2013

domain-system and thus the reduction of currently appearing fragmentation.

Persistent identifiers are of importance for this thesis because subsets of datasets have to be unambiguously identifiable over a long period of time. Thus we will make use of PIDs throughout this thesis.

When it comes to choosing the best concept of PIDs, it cannot be said which solution is the best [34]. Every approach has strengths and weaknesses. Hence it depends on the user's preferences and field of application which PID is chosen, implemented and used. In the decision process the subsequently listed questions should be taken into account.

1. **Administrative commitment:** Is the user aware of future maintenance costs and is he willing to contract other institutions for the maintenance of identifier data?
2. **Existing identifier schemes:** Are identifiers already used for resources? Are they unique and stable?
3. **Available technology and knowledge:** Which identifier approaches are supported by the already existing technologies and which are not?
4. **Users' preference:** Does the user prefer a specific approach?

The intention of all the approaches presented above is the same and the systems have got features in common [34]. Anyways pros and cons of the different systems are discussed controversially and it is still not clear whether every single one of them will have a future or not.

2.4 Summary

In this Section we provided an overview on the topics of data citation and persistent identifiers. In the course of the section covering data citation an introduction was given followed by the presentation of an already existing approach for relational databases. Furthermore we listed scientific research in this field of study of versioning in the context of XML datasets that has been done until present. Basic ideas listed in this chapter are adapted and used by us and thus helped us developing the approaches we will present in this thesis.

The just presented topics are important for this thesis because they build up the basis of the work and investigations that have been done during the writing process of this thesis. Especially defining the basics of data citation as well as presenting already existing approaches to versioning were important.

CHAPTER 3

XML & Description of Used Datasets

In this chapter theoretical basics about the Extensible Markup Language (subsequently abbreviated with XML) will be described. Starting with Section 3.1, a general introduction to XML will be provided. Section 3.2 will discuss the concept of Document Type Descriptors (DTDs) and Section 3.3 will introduce the XML Schema Definition (XSD). In Section 3.4 the concepts of XPath and XQuery will be described briefly. Before this chapter of XML will be concluded with a summary of the just mentioned content in Section 3.6, Section 3.5 will describe the datasets that are used throughout this thesis.

3.1 XML

With the help of XML information is stored in a for computers as well as humans readable way [5]. The specification of XML is set by the World Wide Web Consortium (W3C [1]). Due to the fact that both HTML (Hypertext Markup Language) and XML are closely related to SGML (Standard Generalized Markup Language) [2], the syntax of XML containing tags looks similar to the one of HTML. Open and close tags mark the beginning and the end of elements [22]. Every XML document consists of exactly one root node, which contains all the other nodes [31]. Thus every XML document can be represented as a tree.

The node types we will use throughout this thesis are the following [10] [1]:

1. **Element:** An element node is an information block that is capable of holding an ordered list of child nodes of the type element or text and a set of attributes. It is defined between “ <>” brackets.

[1] Compare `http://www.w3.org/` - Last access: 15-07-2015

[2] XML is a subset of and HTML is an application of SGML. Compare `http://webdesign.about.com/od/sgml/a/how-are-sgml-html-and-xml-related.htm` - Last access: 25-08-2015

2. **Attribute:** An attribute node is a pair of an attribute name and an attribute value. Other than for child elements the order of attribute nodes does not matter in XML documents. All the attribute nodes within a single element node have to have different names but their values do not have to differ.

3. **Text:** A text node is a simple string that can be a child of an element node.

In XML there exist four content models for elements, which are subsequently listed [22].

1. **Element Content:** Elements contain only other elements.

2. **Data Content:** Elements contain only data (text nodes).

3. **Mixed Content:** Elements contain data as well as other elements - in practice XML experts think that using such elements is poor design practice.

4. **Empty:** Elements contain neither elements nor data.

Listing 3.1 examplifies a small part of the dataset that will be described in detail in Section 3.5.2.

Listing 3.1: Example XML: Element Structure

```
<tmSegments>
  <tmSegment>
    <segLabel>A</segLabel>
    <beginIndex>55</beginIndex>
    <endIndex>86</endIndex>
  </tmSegment>
  <tmSegment>
    <segLabel>B</segLabel>
    <beginIndex>124</beginIndex>
    <endIndex>144</endIndex>
  </tmSegment>
  <tmSegment>
    <segLabel>C</segLabel>
    <beginIndex>170</beginIndex>
    <endIndex>201</endIndex>
  </tmSegment>
</tmSegments>
```

In the example shown in Listing 3.1 “tmSegments” represents the identified node, which has got three “tmSegment” child nodes. Each of these nodes again has got three children of the type element. Hence the “tmSegments” and “tmSegment” nodes have got “Element Content” when talking about the just described content models. Taking a look at the node “segLabel” for example shows that its child is a character string and not another element and its content model

is thus “Data Content”.

The concept of attributes will be explained with the help of the in Listing 3.2 depicted root node of the dataset described later in Section 3.5.2 [22]. Attributes provide a solution for adding information to elements. Each attribute consists of both an attribute name and an attribute value. An important fact to mention is that attribute names as well as attribute values are case sensitive. Technically attributes are not strictly necessary as the same information could be provided with the help of subelements, but this concept provides a better and cleaner solution depending on the purpose of the dataset.

Listing 3.2: Example XML: Attributes

```
<mptopo
  createdBy="Stephen White"
  maintainedBy="Craig Snider"
  copyright="Copyright \copyright 1998-2014 Stephen H. White.
     All rights reserved."
  url="http://blanco.biomol.uci.edu/mptopo/mptopoTblXml"
  lastNewMptopoProteinDate="2007-02-07 00:00:00.0"
  lastDatabaseEditDate="2014-08-06 15:23:10.0"
  timeStamp="Wed May 06 14:11:45 PDT 2015">
  [...]
</mptopo>
```

In the example given in Listing 3.2, the element “mptopo” has got seven attributes which contain general information about the dataset. The name of each attribute is given on the left side of the equals sign and the attribute value is given on the right side. Quotation marks form the beginning and the ending of the value definition.

An important feature of XML, which will only be described briefly in this context, is the functionality of namespaces [20]. Given the case that two nodes are named equally, namespaces introduce the possibility to distinguish between them. Thus they are used to disambiguate duplicate element names and group certain elements or attributes. A namespace is identified with the help of a Uniform Resource Identifier (URI) and is normally a HTTP URL such as “http://datypic.com/prod” [69]. However, this URL is not meant to be accessed with the help of a browser but it ensures the uniqueness of the name. A person owning a domain is most likely to have control over it and thus should not use duplicate namespace URIs in the context of a certain domain.

3.2 Document Type Descriptor (DTD)

A Document Type Descriptor (DTD) helps specifying the rules for validating the contents of an XML document [13]. Once the link between the XML file and the DTD is established, the content of the XML document is checked with reference to the rules defined in the DTD.

At this point three of the important benefits of a DTD will be pointed out. First of all, characteristics of a document can be defined in a formal way for later reference. Secondly, extraction and manipulation programs can be written without ever facing unexpected input due to varying structure. The third advantage of using a DTD is that authors and editors can be guided to produce conforming documents.

The two most important declarations concerning DTDs are "ELEMENT" and "ATTLIST". An element declaration is used for defining a new element and for specifying its allowed content. The syntax is as follows: "<!ELEMENT title .. >". The two dots represent the place where the context is defined. The attribute declaration follows the syntax "<!ATTLIST node .. .. >" where "node" identifies the element node the defined attribute belongs to. After the identification of the node, the name and the type of the attribute are stated.

As an example, a part of the DTD of the MPtopo dataset, which will be described in Section 3.5.2, is given in Listing 3.3.

Listing 3.3: DTD Example - external

```
<!ELEMENT mptopo (caption,groups*)>
<!ATTLIST mptopo createdBy CDATA #REQUIRED>
<!ATTLIST mptopo maintainedBy CDATA #REQUIRED>
<!ATTLIST mptopo copyright CDATA #REQUIRED>
<!ATTLIST mptopo url CDATA #REQUIRED>
<!ATTLIST mptopo lastNewMptopoProteinDate CDATA #REQUIRED>
<!ATTLIST mptopo lastDatabaseEditDate CDATA #REQUIRED>
<!ATTLIST mptopo timeStamp CDATA #REQUIRED>
<!ELEMENT caption (#PCDATA)>
<!ELEMENT groups (group*)>
[...]
```

In the example above it can be seen that the element node named "mptopo" is the root node. It holds one "caption" element and zero to an infinity of instances of "groups" element nodes as children. Additionally seven required attributes, all of type "CDATA" (CDATA stands for "character data" and is used for text that is not parsed by the XML parser [3]), belong to the "mptopo" element. After the definition of the root node, the structure of its children is listed. The "caption" node is an element containing only data and thus its content model is "Data Content" (compare the content models for elements in Section 3.1). The other child, "groups", is an "Element Content" node due to the fact that it contains only zero to an infinity of "group" elements. These nodes again consist of two children, which are not going to be described any further because the definition of these nodes follows the same concept as the already stated ones.

A DTD can either be defined right inside an XML file (internal DTD) or in an external file (external DTD). If the DTD is declared in an external file, the .dtd file follows the in Listing 3.3

[3] Compare `http://www.w3schools.com/xml/xml_cdata.asp` - Last access: 17-08-2015

depicted structure and the XML file needs to reference to this file using a “<!DOCTYPE>” tag. If the DTD is declared inside an XML file, it is wrapped inside the <!DOCTYPE>” definition. Thus the DTD starts with the tag “<!DOCTYPE” followed by the name of the document type. It has to be taken into account that the name of the document type has to be equal to the name of the root node of the XML file. The DTD is then defined between square brackets “[..]”. After the definition of the DTD, the normal XML file is listed.

To make the structure of an internal DTD clearer, Listing 3.4 provides a brief example.

Listing 3.4: DTD Example - internal

```
<!DOCTYPE mptopo [
  <!ELEMENT mptopo (caption,groups*)>
  <!ATTLIST mptopo createdBy CDATA #REQUIRED>
  [...]
]>
```

3.3 XML Schema Definition (XSD)

In the previous section it was shown that the structure of XML documents can be described with the help of a DTD. Besides DTDs, the structure and thus legal building blocks of XML documents can also be defined with the help of XML schemas [13]. Such models are backward-compatible with DTDs and thus it is possible to convert a DTD into an XML schema. Converting an XML schema into a DTD is not always possible as not all the features of XSDs are supported by DTDs.

As an example, a part of the XSD of the MPtopo dataset, which will be described in Section 3.5.2, is presented in Listing 3.5.

Listing 3.5: XSD Example

```
<xs:element name="mptopo">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="caption"/>
      <xs:element ref="groups" minOccurs="0" maxOccurs="unbounded
          "/>
    </xs:sequence>
    <xs:attribute name="createdBy" type="xs:string" use="
       required"/>
    <xs:attribute name="maintainedBy" type="xs:string" use="
       required"/>
    <xs:attribute name="copyright" type="xs:string" use="
       required"/>
```

```
    <xs:attribute name="url" type="xs:string" use="required"/>
    <xs:attribute name="lastNewMptopoProteinDate" type="xs:
       string" use="required"/>
    <xs:attribute name="lastDatabaseEditDate" type="xs:string"
       use="required"/>
    <xs:attribute name="timeStamp" type="xs:string" use="
       required"/>
  </xs:complexType>
</xs:element>
<xs:element name="caption" type="xs:string"/>
<xs:element name="groups">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="group" minOccurs="0" maxOccurs="unbounded"
         />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

As the example in Figure 3.5 shows, each node in an XML document is defined with the help of an “<element>” or an “<attribute>” block. The nodes’ names are defined with the help of “name” and the type of the respective node is defined by “type”. If, for example, an element node is defined by the information that its type is a string (type=“xs:string”), then the node itself is an element having a text node as child. The attributes “minOccurs” and “maxOccurs” indicate how often a certain element node is allowed to appear and the attribute “use” indicates whether an attribute’s existence is compulsory or not.

Elements can be defined at any position and referenced at another. An example for this is the element “caption”, which is a child element of the “mptopo” node. The first part of <xs:element ref=“caption”/> indicates that this node is an element and the ref=“caption” part references to the definition of the “caption” element.

3.4 XPath and XQuery

The two concepts XPath and XQuery help navigating through XML documents and finding and retrieving required information in XML datasets [13]. Furthermore objects, which do not have a unique ID assigned, can be identified. This means that the XML Path Language (XPath) allows expressions to identify subtrees within an XML document [31]. Every XML tree consists of exactly one root node, which contains all the other nodes. XPath serves the purpose of selecting nodes and sets of nodes out of this tree.

The most important feature of XPath are so called location paths, which identify a certain set of nodes in a document. Thus the set can contain one to several nodes but may also be empty.

The easiest location path is the one selecting the root node (with the help of the expression "/*"). If child nodes of the actual context node should be selected, a hierarchy aparted by forward slashes can be given. To address attributes the operator "@" is used.

An example for an XPath expression is depicted in Listing 3.6. In this example every "mptopoProtein" node that has got a "proteinName" child node with the content "LHC-II" is identified. After having identified all the nodes (in the case of the MPtopo dataset only one), all the child nodes "tmSegments" are selected and printed out (again, only one node). The two slashes at the start of the query signalise that not an absolute path is provided and that thus the whole XML tree is searched for the wanted node(s) because of the fact that the expression of two slashes is defined as "descendant or self" [37].

Listing 3.6: Sample Query Identifying "tmSegments" Element Nodes

```
//mptopoProtein[proteinName = "LHC-II"]/tmSegments
```

The only example in the domain of the MPtopo dataset including attributes is the root node of the document as it is the only element node containing attributes. The example given in Listing 3.7 identifies every "mptopo" element node that has got an attribute named "createdBy" and of which the attribute value is "Stephen White". After the element nodes have been identified, all the child elements named "caption" are selected and printed out. Due to the fact that the "mptopo" node is the root node, one slash would be sufficient because a single slash at the start of the query signals an absolute path.

Listing 3.7: Sample Query with the Usage of Attribute

```
//mptopo[@createdBy = "Stephen White"]/caption
```

XQuery makes selections, reorganisations and transformations of XML data elements of interest possible [24]. The results can be returned in a structure of the user's choice [69]. The XQuery language is based on XPath and its syntax is thus similar to the one of XPath.

A very important concept of XQuery is the so called FLWOR expression [37]. Such an expression consists of the parts "for", "let", "where", "order" and "return" and is similar to the SELECT-FROM-WHERE statement in SQL. Tuples are built in the "for" and "let" clauses, the "where" and "order" clauses filter and order and the "return" clause prepares the output of the whole FLWOR expression. A simple example showing the functionality of the FLWOR concept is provided below in Listing 3.8.

Listing 3.8: FLWOR Query Example

```
for $node in //tmSegment[segLabel = "A"] \\
let $beginIndex := $node/beginIndex \\
where $beginIndex/text() < 50 \\
return replace value of node $beginIndex/text() with -1
```

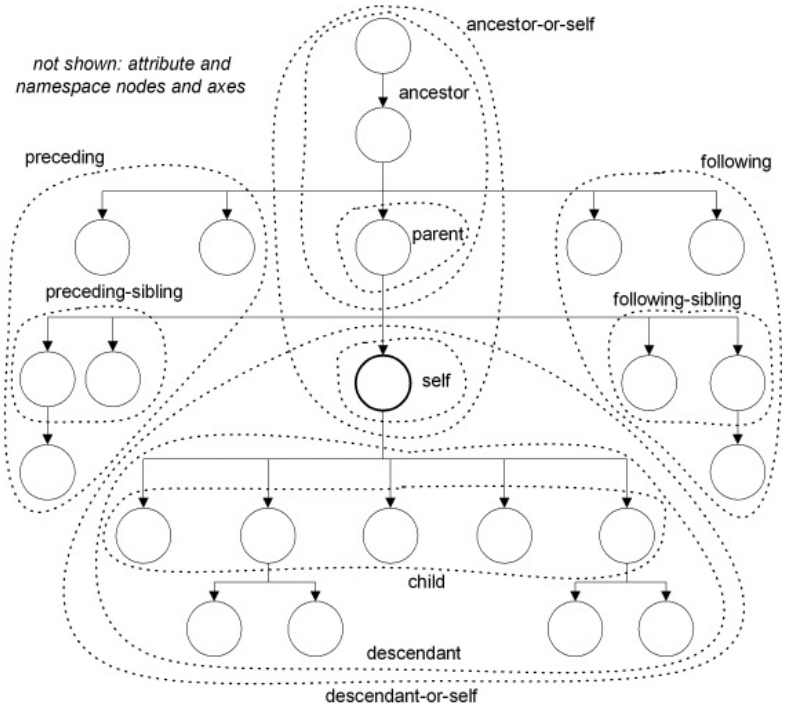

Figure 3.1: Concept of Different XPath Axes [6]

In this example shown above (Listing 3.8) a certain subset of nodes is selected and iterated through. For each node the variable “beginIndex” is set and afterwards the respective text child node of each “beginIndex” node is checked whether its value is smaller than 50 or not. If yes, the value is replaced with the value -1.

A last piece information that is provided at this point, is the concept of axes XPath uses. These axes show how the rest of the dataset is related to the current node. Figure 3.1 depicts theses axes [6]. The current node is positioned in the center of the figure and the circles imply which nodes can be addressed with the help of which axis. So it can be seen that for example the current node itself can be addressed with the help of the “self”-axis and its children can be addressed by using the “child”-axis. If one wants to receive all the nodes on the path from the root node to the current node, the “ancestor”-axis can be used. With the two axes “attribute” and “namespace”, which are not depicted in Figure 3.1, all the attributes / all the namespace nodes of the current node can be addressed.

Concluding this section it can be said that XPath addresses nodes and thus parts of an XML document. However, XPath was not designed for activities such as introducing variables or namespace bindings, calculating the maximum/minimum of a set of numbers, etc. This is where XQuery comes into play, providing the features to query a document or dataset adequately.

3.5 Description of the Used Datasets

In this section we will introduce the datasets that are used throughout this thesis. The datasets that will be presented in the first part, Section 3.5.1, are artificially generated ones and thus do not contain real world data. The MPtopo dataset that will be presented in Section 3.5.2 refers to a real dataset. The artificially generated datasets will serve the purpose of measuring the performance of our implemented solutions and are thus benchmark datasets and the MPtopo dataset is used as an example to illustrate the approaches.

3.5.1 Generated Datasets

In order to analyse the performance of the two approaches presented by us, we created two datasets that differ in their hierarchy. The first dataset shows a rather deep hierarchy having 13 levels and the second one was created as a flat hierarchy dataset showing five hierarchical levels. Because of the fact that pieces of information are mostly stored in leaf text nodes, we ensured that both datasets have got the same amount of text nodes, namely 531 441.

Both datasets listed in this section were created with the help of the DBMS "BaseX". The query provided in Listing 3.9 was run iteratively for both datasets to insert a certain number of child nodes into every element node of the lowest hierarchical level. For the deep dataset three children were inserted per element node ("for 1 to 3" loop) and 27 for the flat hierarchy dataset ("for 1 to 27" loop). After the execution, this query was edited in order to be able to insert "level2" elements into every "level1" element node. The query depicted in Listing 3.10 was used at the end of the creation of the whole hierarchy in order to insert all the text leaf nodes. To help understanding the structure of the datasets, Figure 3.2 depicts the first three hierarchical levels of the dataset with deep hierarchy. It can be seen that every element node contains three attributes named "id", "attr" and "level". The "id" as well as the "level" attributes are used to address elements and are thus never edited in the course of our performance tests.

Listing 3.9: Query for the Generation of the Datasets - Element Nodes

```
for $node in //level0 return (
for $i in 1 to 3 return (
insert node <level1 id="{$i}" attr="att"/> into $node
) )
```

Listing 3.10: Query for the Generation of the Datasets - Text Nodes

```
for $node in //level12 return (
insert node "textNode" into $node
)
```

After the generation of the two datasets, both of them were edited with the help of the DBMS "BaseX" again. Listing 3.11 depicts the query that was executed in order to insert the timestamps "xmlDB_inserted" and "xmlDB_deleted" into all the nodes of type "element" (needed

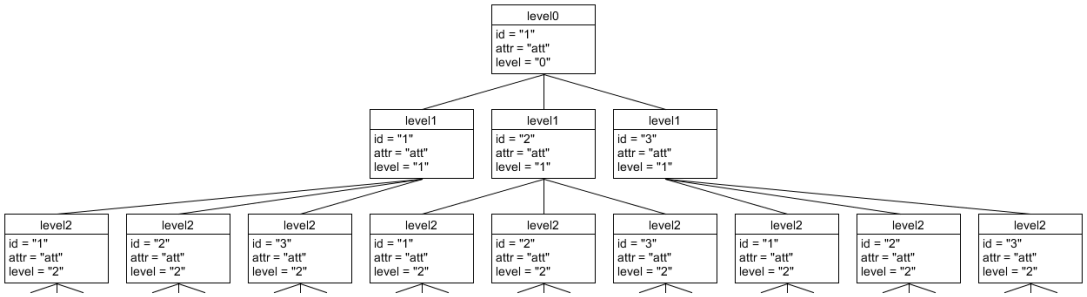

Figure 3.2: Artificial Dataset - Example Deep Hierarchy

Table 3.1: Specifications Generated Datasets (Deep Hierarchy)

Information	Normal	Approach I	Approach II
Number of Nodes in total	3 720 086	5 314 408	3 720 087
Number of document nodes	1	1	1
Number of element nodes	797 161	797 161	797 161
Number of attribute nodes	2 391 483	3 985 805	2 391 484
Number of text nodes	531 441	531 441	531 441
Size of .xml File	67.29 MB	126.28 MB	67.30 MB

for Approach I - “Branch Copy”) and Listing 3.12 was used for the insertion of the timestamp “xmlDB_inserted” into the root node (needed for Approach II - “Parent-Child”).

Listing 3.11: Insertion of Versioning Timestamps for History DB - Approach I - “Branch Copy”

```
for $node in //* return (
insert node attribute xmlDB_deleted {"null"} into $node,
insert node attribute xmlDB_inserted {fn:current-dateTime()}
   into $node
)
```

Listing 3.12: Insertion of Versioning Timestamps for History DB - Approach II - “Parent-Child”

```
insert node attribute xmlDB_inserted {fn:current-dateTime()}
   into /*
```

The specifications of the datasets created by us are listed in Table 3.1 for deep hierarchy and in Table 3.2 for flat hierarchy. It is important to mention that although the datasets contain the same number of text nodes, their amount of element and attribute nodes and thus their size differs. This is the case because of the fact that their structure and hierarchy is different. We set the number of text nodes to 541 441 because it is a multiple of 27 and 3. With these numbers being the amount of children of type element every single element node has got, we were able to construct two datasets with a different hierarchy both having a size between 30 and 70 MB.

Table 3.2: Specifications Generated Datasets (Flat Hierarchy)

Information	Normal	Approach I	Approach II
Number of Nodes in total	2 738 966	3 842 728	2 738 967
Number of document nodes	1	1	1
Number of element nodes	551 881	551 881	551 881
Number of attribute nodes	1 655 643	2 759 405	1 655 644
Number of text nodes	531 441	531 441	531 441
Size of .xml File	33.90 MB	74.74 MB	33.90 MB

3.5.2 MPtopo Dataset

MPtopo is a curated database of membrane proteins [4] with experimentally validated transmembrane [5] segments [6] [36]. The database was created because it is needed for developing new computational tools for the prediction of membrane protein (MP) structure. MPtopo is freely available on the internet at `http://blanco.biomol.uci.edu/mptopo` and can be directly downloaded or queried using a Java Applet.

The MPtopo dataset is provided by the Stephen White laboratory at UC Irvine [7] [36]. It is is a subset of the mysql database named MPstruc [8] and made available as XML. Currently the MPtopo dataset contains 165 different proteins (node name: mptopoProtein) and 949 TM segments (node name: tmSegment). The XML file's size is roughly 500 KB and contains 14581 nodes (8965 of them being elements) in total.

Taking a look at the DTD and the dataset, it can be seen that there is a root node with the name "mptopo", which contains the required attributes "createdBy", "maintainedBy", "copyright", "url", "lastNewMptopoProteinDate", "lastDatabaseEditDate" and "timeStamp". The last three attributes are highlighted, as "lastNewMptopoProteinDate" and "lastDatabaseEditDate" are timestamps showing when the last protein was added to the database and when the database was edited for the last time. The attribute with the name "timeStamp" shows when the dataset was downloaded. Thus it can be seen that the last protein was added on 2007-02-07, the database was last edited on 2014-08-06 and that the database was downloaded on 2015-05-06 14:11:45 PDT. Table 3.3 depicts the specifications of the dataset.

3.6 Summary

In this chapter we provided an overview on the basics of XML. DTDs and XSDs were described and explained on the basis of the examples provided above. Accessing and editing XML datasets is made possible with the help of the two technologies XPath and XQuery. Both of them are im-

[4] "A protein that is attached to, or associated with, a biological membrane." - `http://www.biology-online.org/dictionary/Membrane_protein` - Last access: 21-08-2015

[5] Transmembrane means "through or across a membrane". - `http://www.biology-online.org/dictionary/Transmembrane` - Last access: 21-08-2015

[6] Compare `http://blanco.biomol.uci.edu/mptopo/` - Last access: 28-08-2015

[7] Compare `http://blanco.biomol.uci.edu/index.shtml` - Last access: 03-05-2014

[8] Compare `http://blanco.biomol.uci.edu/mpstruc/` - Last access: 03-05-2014

Table 3.3: Specifications MPtopo Dataset

Information	Values
Number of Nodes in total	15 302
Number of document nodes	1
Number of element nodes	8 965
Number of attribute nodes	7
Number of text nodes	6 329
Size of .xml File	506 KB

portant for this thesis as editing and retrieving subsets of certain datasets are needed in order to achieve our set goals.

An important section of this chapter was the description of the datasets we use in this thesis. Artificially generated datasets as well as a dataset containing real world data were presented and described.

CHAPTER 4

Databases for XML

In this chapter we will provide an insight into the topic of databases for XML. After a short overview of database systems in general, Section 4.1 will briefly highlight the concept of relational databases as well as enabled XML databases. Section 4.2 will provide an insight into native XML databases. The two Subsections 4.2.1 and 4.2.2 will introduce two programs which will be used in the course of this thesis, namely "BaseX" and "eXist-db". Afterwards Section 4.3 will provide a summary of this chapter.

In general a database system consists of two components, namely the database itself and the database management system (DBMS) [28]. The database is a collection of logically related data which can be recorded. A database management system is a set of programs that is used for defining, creating, maintaining and manipulating a certain database.

Generally it can be said that the first component, a database, is helpful to consolidate and control certain data centrally. Subsequently advantages of databases are listed [38].

1. Redundancy can be reduced [1]
2. Inconsistency can be avoided
3. Data can be shared
4. Standards & Security can be enforced
5. Integrity can be maintained

[1] Redundancy denotes the fact that copies of data are stored at different places in the dataset [53]. It can be reduced with the help of the process of normalization. However, as the topic of normalization mainly relates to RDBMS, this advantage is not the centre of attention of this thesis.

The second component of a database system, the database management system, is a set of programs used to control and manage the use of data [51]. Thus an interface between the database and its users is provided. Consecutively major roles of a DBMS are listed.

1. Data structures for data storage are defined
2. Mechanisms for data access and manipulation are provided
3. System integrity is maintained
4. Safety and security measures for data are provided
5. Mechanisms for data sharing among other users are provided

When it comes to XML data being stored in databases, efficiency is an important criterion [24]. Creating an in-memory document object model (DOM) becomes impractical when dealing with growing documents. Thus efficiency of searching is important. Another factor to be considered is the reliability of the database because a database with high downtimes is unacceptable in a production setting. If XML data is stored in a relational form, it also has to be considered how fast data can be converted to XML and back again. Anyway, this requirement strongly depends on the purpose of the database.

XML data count as semi-structured data which means that information that is normally associated with a schema, is stored within the data and can therefore be called "self-describing" [2]. The structure of XML documents does not correspond to schema models of any relational database management system and so transformations had to be found in order to be able to store them in RDBMS [17]. As another possibility to store XML data in databases, so called native XML databases have been implemented.

4.1 Relational Database Management Systems / Enabled XML Databases

Although the definition is far from being complete, a relational database can be seen as a collection of data, which are represented as two-dimensional tables [58]. Columns of the tables represent component fields of the record and the rows can be seen as instances of a record.

Enabled XML databases are non native XML databases, which are capable of transforming data from the XML structure to their own internal data structure and the other way round [12]. For storing XML documents e.g. in relational databases hierarchically, tree-structures have to be described with the help of relations [17]. Thus XML document schemas have to be translated into the relational schema before accessing the according tables. At the same time XML query

[2]Compare `http://homepages.inf.ed.ac.uk/opb/papers/PODS1997a.pdf` - Last access: 10-02-2014

tmSegmentID	segLabel	beginIndex	endIndex	tmSegments
1	A	55	86	187
2	B	124	144	187
3	C	170	201	187

Figure 4.1: Example of a Table in a Relational Database Management System

languages have to be translated into SQL to access the data.

As an example of XML data being stored in a relational database, the subset of XML data, which has already been presented in Listing 3.1 in Section 3.1, is used. Figure 4.1 shows an example of how these data could be fit into a table. Each row represents a single "tmSegment". The attribute "tmSegmentID" was additionally added as primary key and the attribute "tmSegments" was added as a foreign key showing the "tmSegments"-entry the respective tmSegment belongs to. As it can be seen the hierarchy was completely resolved and is represented with the help of IDs and references to them in order to fit the data in single rows.

Early implementations of databases, which were able to deal with XML, involved storing XML files in table colums [69]. Query access was provided to these colums. Nowadays the line between native XML databases and relational databases is blurring and thus more and more relational databases offer the feature of storing XML data in a native way. The authors of [11] stated that enabled XML databases are useful if relational data have to be published in XML format or data in XML format have to be imported into an existing relational database. The downside of enabled XML databases is that they do discard information like the sibling order for example. Due to the fact that the focus of this thesis lies of native XML databases, enabled databases will not be described in detail.

Concerning the usage of native XML databases it can be said that they shine when it comes to data that do not easily fit a relational but rather the XML data model [11]. Common use-cases of native XML databases are e.g. handling very quickly evolving schemas or dealing with hierarchical data. The reason for the usage of native XML databases are the same as the ones for other database types. The management of data becomes easier, concurrent access is managed by the application and query performance is improved.

4.2 Native XML Databases

The approach of native XML database management systems (XDBMS) was designed to exclusively manage XML data [30]. In such databases XML data are stored persistently in their tree-like structure. Thus relationships like the parent-child relation or the siblings relation are reflected by the internal storage structure. Hence it can be said that the term "native" refers to

the fact that the data store application keeps the XML syntax intact [29].

Native XML databases are often used for narrative data and such data that are less predictable than what would normally be stored in relational databases [69]. This is the case because XML databases do not have to have a fixed structure and thus can be schemaless allowing all kind of nodes to be inserted. Native XML database management systems provide traditional features such as data storage, indexing, loading, querying and extracting as well as backup and recovery operations.

The author of [65] defines a native XML DBMS as follows.

1. The XDBMS defines a model for an XML document.
2. The XDBMS has got an XML document as its fundamental unit of storage just as a relational databases fundament is a row in a table.
3. The XDBMS does not need a particular underlying physical storage model.

The following two Subsections 4.2.1 and 4.2.2 will provide an overview and will introduce two XDBMS, namely "BaseX" and "eXist-db", which will be used in the empirical part and thus throughout this thesis.

4.2.1 BaseX

"BaseX" is a scalable and high-performance XML database engine that focuses on storing, querying, and visualizing large XML and JSON (JavaScript Object Notation) documents and collections [3] [4]. It is an open source software project and can be downloaded via GitHub [5]. Besides the developer version, pre-built versions can also be downloaded on the website [6]. Information about the software is available at the "BaseX" homepage [7]. In this thesis version 8.2.3 is used.

Besides a Client/Server architecture, which is able to handle concurrent read and write operations of multiple users, a visual frontend is provided. With the help of that frontend users are able to interactively explore data and perform queries. Due to the fact that the parser we will present in order to enable versioning of datasets is not capable of dealing with concurrent requests, we insert a locking attribute into the dataset in order to imply that the dataset is currently used by another process or user.

"BaseX" provides implementations of XPath and XQuery, which we have already described in Section 3.4, as well as the XQuery Update Facility (XQUF) [8]. With the help of the first two

[3] Compare `http://basex.org/products/` - Last access: 15-09-2015
[4] Compare `http://docs.basex.org/wiki/Main_Page` - Last access: 15-09-2015
[5] Compare `https://github.com/BaseXdb` - Last access: 15-09-2015
[6] Compare `http://basex.org/products/download/all-downloads/` - Last access: 15-09-2015
[7] Compare `http://basex.org/` - Last access: 15-09-2015
[8] Compare `http://docs.basex.org/wiki/Updates` - Last access: 15-09-2015

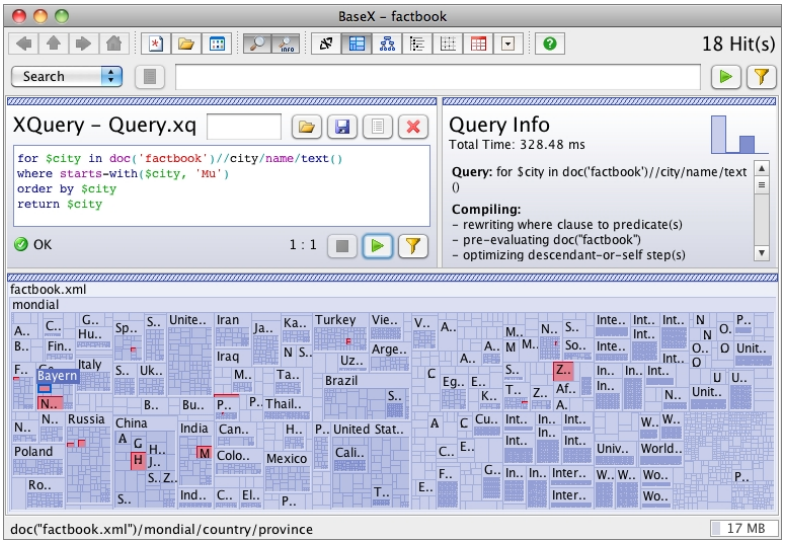

Figure 4.2: GUI of the BaseX Database Management System "BaseX" [9]

concepts data can be addressed and queried and XQUF helps making persistent changes to the dataset by for example inserting or deleting nodes. Although XQUF includes a function named "transform" that is supported by "BaseX", it is not included in this thesis as it is not supported by the DBMS "eXist-db".

To provide an insight into the tool, Figure 4.2 shows the graphical user interface (GUI) and thus the visual frontend of "BaseX". In the upper left corner queries can be entered and on the right side information about just executed queries are displayed. On the bottom, the database or the respective subset identified by the user is visualised as a map. Concerning the visualisation, a lot of different types like e.g. "Map", "Tree" or "Folder" can be chosen.

4.2.2 eXist-db

"eXist-db" is an open source native XML database management system that enables users to index and query XML resources [10]. The software project started as an open source community project that was built on the contribution of volunteers [11]. After some time, members of the development team formed the company "eXist Solutions" [12] in order to be able to offer professional support and solutions for "eXist-db".

[9] Compare `http://basex.org/products/gui/` - Last access: 16-05-2014

[10] Compare `http://existsolutions.com/existdb.xml#d601668e115` - Last access: 15-09-2015

[11] Compare `http://www.exist-db.org/exist/apps/doc/getting-help.xml` - Last access: 15-09-2015

[12] Compare `http://www.existsolutions.com/` - Last access: 15-09-2015

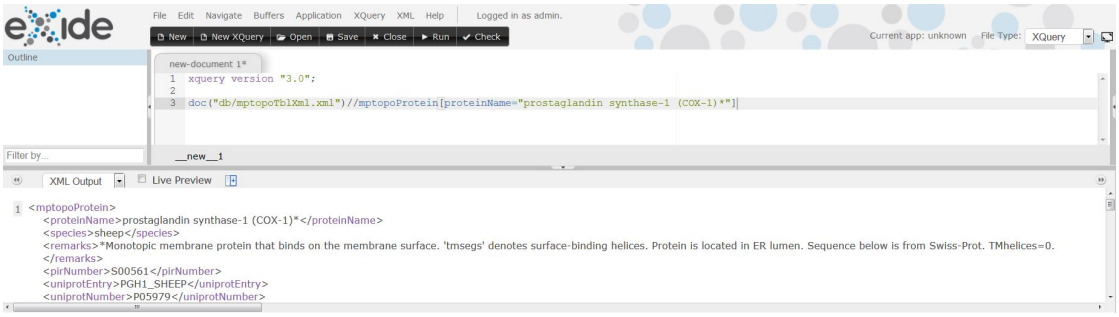

Figure 4.3: eXide of the Database Management System “eXist-db”

The latest versions of the software can be downloaded directly from the homepage of the software [13]. The open source code of “eXist-db” is made available and can be downloaded from GitHub [14]. In this thesis the at the time of writing latest stable version 2.2 is used.

Like the DBMS “BaseX” “eXist-db” supports the concepts of XPath and XQuery as well as the concept of the XQuery Update Facility. With these technologies being supported, the two DBMS “BaseX” and “eXist-db” perfectly fit the purpose of this thesis.

To provide an idea of the looking of the central graphical user interface (GUI) of “eXist-db”, Figure 4.3 shows the so called “eXide - XQuery IDE”. The eXide provides an interface to run queries onto datasets in the database. In the top half of the screen a query can be entered and the result is displayed at the bottom half of the screen. In this case the user queried a mptopoProtein that has “prostaglandin synthase-1 (COX-1)*” as proteinName.

4.2.3 Differences between “BaseX” & “eXist-db” CRUD Operations

Due to the fact that the two database management systems we use in this thesis react differently to certain inputs, we will list the differences in this section. All the information given in this section was collected through experiments. The different cases (combinations of the type of the user entered expression and the type of the addressed node - two examples for the “insert” operation: “insert node TEXT into ELEMENT” and “insert node ATTRIBUTE into ELEMENT”) for each CRUD operation have been run through with sample queries in both database management systems.

While analysing the different CRUD operations and all their different combinations of node types we figured out that not all the cases are possible for the database management systems. The combinations yielding errors for at least one DBMS are disabled by the parser in order to prevent error cases. The disabling mechanism is a check for the types of all the in the database

[13] Compare `http://exist-db.org/exist/apps/homepage/index.html` - Last access: 15-09-2015
[14] Compare `https://github.com/eXist-db/exist` - Last access: 15-09-2015

addressed nodes against the type of the user entered expression. If the combination is a possible one, the query is executed - in case of a disabled combination an error message is printed to the user.

The subsequently following Sections 4.2.3.1 - 4.2.3.5 will describe the five CRUD operations and Section 4.2.3.6 will summarise them.

4.2.3.1 Insert

For the "insert" operation it is possible to insert element, attribute and text nodes into elements. Inserting any kind of node into attribute or text nodes is not possible in both of the DBMS and therefore also disabled in the parser presented in this thesis. The two Listings 4.1 and 4.2 depict the query syntax of the "insert" operation for the DBMS "BaseX" and "eXist-db".

Listing 4.1: Query Syntax of the "Insert" Operation Query Syntax - DBMS: BaseX

```
insert node [Expression] into [Target]
```

Listing 4.2: Query Syntax of the "Insert" Operation Query Syntax - DBMS: eXist-db

```
update insert [Expression] into [Target]
```

4.2.3.2 Delete

The deletion of all the three types of nodes that we consider in this thesis, namely element, attribute and text, is possible. The two Listings 4.3 and 4.4 depict the query syntax of the "delete" operation for the DBMS "BaseX" and "eXist-db".

Listing 4.3: Query Syntax of the "Delete" Operation Query Syntax - DBMS: BaseX

```
delete node [Target]
```

Listing 4.4: Query Syntax of the "Delete" Operation Query Syntax - DBMS: eXist-db

```
update delete [Target]
```

4.2.3.3 Replace

Figure 4.1 depicts the different cases of the "replace" operation. In this table the terms "element", "text" and "attribute" refer to the respective node types. So "element" stands for an element node and so on. On the x-axis the two database management systems, namely "BaseX" and "eXist-db" and the respective queries are listed. The variables [Type 1] and [Type 2] are definable parameters. The values of these parameters are defined on the y-axis. The tables shows all the possible combinations of the three node types "element node", "attribute node" and "text node". Each column represents the behaviour of the DBMS when a query is executed with the respective combination of node types.

Table 4.1: Comparison of Replace-Operation

Type 1	Type 2	"BaseX" replace node [Type 1] with [Type 2]	"eXist-db" update replace [Type 1] with [Type 2]	Comparison
element	element	Old element is replaced with new element	Old element is replaced with new element	Same
	attribute	ERROR: Replacing nodes must be no attribute nodes	Old element is deleted and new attribute is inserted into parent node of old element	Different
	text	Old element is replaced with new text	Old element is replaced with new text	Same
attribute	element	ERROR: Replacing nodes must be attribute nodes	old value of attribute is deleted and nothing inserted EXCEPTION: new element has got a text as child -> text is set as new value of attribute	Different
	attribute	Old attribute is replaced with new attribute	Old attribute is replaced with new attribute	Same
	text	ERROR: Replacing nodes must be attribute nodes	old value of attribute is replaced with new text	Different
text	element	Old text is replaced with new element	old text is deleted and nothing inserted EXCEPTION: new element has got a text as child -> old text is replaced with new text	Different
	attribute	ERROR: Replacing nodes must be no attribute nodes	Old value of text is replaced with the value of the new attribute	Different
	text	Old text is replaced with new text	Old text is replaced with new text	Same

The last column in each table is named "Comparison" and represents the comparison between the behaviour of the two database management systems for each of the different combinations of node types. The columns are filled with "Same", when both of the DBMS show the exact same behaviour and "Different" if the outcome is different. The colouring of the cells of the tables indicates whether the parser presented by us supports the respective combination of node types for the respective DBMS. A grey coloured cell means that this operation is disabled and will thus yield an error. Some cells represent a combination of node types that yield errors but have an exceptional case in which these errors are not thrown and updates are performed (tagged with an "EXCEPTION:" string). We disabled these specific cases on purpose because we identified that they do not make sense in practice.

When looking at the table it can be seen that we disabled the possibilities of replacing an element node with an attribute node as well as replacing a text node with an attribute node for the DBMS "eXist-db" because we doubt that these cases find application in practice. The same applies for the replacement of an element with a text (both DBMS) and the replacement of a text with an element (both DBMS as well) because we think that this will most probably lead to a violation of our assumption that element nodes only contain either one or more element nodes or a single text node as child/children. The last behaviour we disabled for the "replace" operation is the replacement of an attribute with a text node for the DBMS "eXist-db". This was done because of the fact that the same result is achieved with the "replace value" operation when the value of an attribute node is replaced with a text node.

4.2.3.4 Replace Value

For explaining the different cases of the "replace value" operation we use the same type of table that we have already used for the "replace" operation in Section 4.2.3.3. Figure 4.2 depicts the comparison of the "replace value" ("BaseX") or "value" ("eXist-db") operation. Like for the

Table 4.2: Comparison of Replace Value-Operation

Type 1	Type 2	"BaseX" replace value of node [Type 1] with [Type 2]	"eXist-db" update value [Type 1] with [Type 2]	Comparison
element	element	All children of old element are deleted and nothing inserted EXCEPTION: new element has got a text as child -> text is set as only child	all children of element are deleted new element is inserted as only child	Different
	attribute	all children of old element are deleted text (value of new attribute) is inserted as only child	all children of element are deleted new attribute is inserted into parent node	Different
	text	all children of old element are deleted text is inserted as only child	all children of element are deleted new text is inserted as only child	Same
attribute	element	old value of attribute is deleted and nothing inserted EXCEPTION: new element has got a text as child -> text is set as new value of attribute	old value of attribute is deleted and nothing inserted EXCEPTION: new element has got a text as child -> text is set as new value of attribute	Same
	attribute	old value of attribute is replaced with the value of the new attribute	old value of attribute is replaced with the value of the new attribute	Same
	text	old value of attribute is replaced with new text	old value of attribute is replaced with new text	Same
text	element	old text is deleted and nothing inserted EXCEPTION: new element has got a text as child -> old text is replaced with new text	old text is deleted and nothing inserted EXCEPTION: new element has got a text as child -> old text is replaced with new text	Same
	attribute	old text is replaced with value of new attribute	old text is replaced with value of new attribute	Same
	text	old text is replaced with new text	old text is replaced with new text	Same

"replace" operation, we disabled the possibility of replacing the value of an element node with an attribute node for both DBMS. A last operation we disabled is the replacing of the value of an element node with another element for the DBMS "eXist-db" as we wanted to standardise the behaviour of the parser and this operation yields an error when using the DBMS "BaseX".

4.2.3.5 Rename

For the "rename" operation it is possible to rename element and attribute nodes. The new name has to be entered as a node of type text. This means that the two combinations "rename element as text" and "rename attribute as text" are enabled. Entering the new name as attribute is possible as well because the old element/attribute node is renamed as the value of the new attribute. However, this option was disabled because it does not make any sense in our eyes. The two Listings 4.5 and 4.6 depict the query syntax of the "rename" operation for the DBMS "BaseX" and "eXist-db".

Listing 4.5: Query Syntax of the "Rename" Operation - DBMS: BaseX

```
rename node [Target] as [Expression]
```

Listing 4.6: Query Syntax of the "Rename" Operation - DBMS: eXist-db

```
update rename [Target] as [Expression]
```

4.2.3.6 Summary of Operations being Enabled by the Parser

Table 4.3 provides an overview over all the allowed CRUD operations and thus summarises what operations are enabled by our parser.

Table 4.3: CRUD Operations Enabled for our Versioning Concepts

Insert	Element into Element Attribute into Element Text into Element
Delete	Element Attribute Text
Replace	Element with Element Attribute with Attribute Text with Text
Replace Value	Element with Text Attribute with Text Text with Text
Rename	Element as Text Attribute as Text

4.3 Summary

In this chapter the topic of XML databases was highlighted. To point out the differences between native and enabled XML databases, the following three points identified by the author of [56] are listed.

1. Native XML databases can preserve physical structures like DTDs, CDATA sections or comments (XML-enabled databases can do that in theory but it is not common in practice [29].).

2. Native XML databases are able to store XML documents without knowing their schema or DTD (XML-enabled databases could create schemas on their own but this is impractical, especially when the documents are schemaless [15] [29].).

3. Native XML databases offer only XML-based interfaces like XPath to access the database.

Nowadays most of the databases are XML enabled but there are also native XML databases [56]. Both types of databases serve a distinct purpose. A relational database management system could for example be used for dealing with a large amount of data and the native XML database could be used for storing the most frequently used tables and datasets of the RDBMS.

[15] If neither a DTD nor an XML Schema is given, the document is schemaless and therefore flexible.

CHAPTER 5

Devising a Versioning and Subsetting Solution

The first part of this chapter, Section 5.1, will cover details of the design process that was run through in order to achieve the results presented in this thesis. The following two Sections 5.2 and 5.3 will explain the two presented and implemented approaches. After that three approach independent operations will be highlighted in Section 5.4. After having defined approach specific and approach independent operations, Section 5.6 will describe our general idea of data citation in the context of native XML databases and afterwards Section 5.5 will point out requirements and what is needed in order to meet them. Concluding this chapter, Section 5.7 will provide a short summary.

5.1 Design

As it has already been stated in Section 2.1.1, we need two components in order to be able to make subsets of datasets citable - on the one hand versioning of the dataset and on the other hand a mechanism that allows to archive queries that derive subsets and additionally makes their re-execution possible. Concerning the archivation of queries we use the concept of a query store, which we have already introduced in Section 2.1.1. We will highlight details on our implementation of a query store in Section 5.6.2. Talking about the versioning of datasets we developed two approaches, which will be described in the Sections 5.2 and 5.3. For making the two approaches work we implemented a parser that acts as a middleware between the user and the DBMS. Every query that is entered by the user is parsed and analysed and the parser invokes and ensures a proper versioning. For further descriptions please check Section 5.6.

Before we implemented both approaches to the topic of versioning and a parser that makes it possible to use these two approaches, we went through a process of conceptual thinking. In this

section we will provide approaches we considered in order to achieve our goals but that were turned down. Furthermore we will explain why they were not suitable. Section 5.1.1 will deal with turned down approaches concerning the functioning of the parser and the processing of queries entered by the user and Section 5.1.2 will describe turned down approaches concerning versioning of datasets.

5.1.1 Turned Down Designs for Functioning of the Parser

The first idea was based on reworking the query string that is sent to the database management system with the help of string operations. Only focusing on the DBMS "BaseX" we wanted to intercept the string before it is processed on the server side of the DBMS. If for example a "delete" operation is to be executed on an element node, it can be rewritten in order to just insert an attribute that implies that this element was disabled. So a "delete" operation can be turned into an "insert" operation by replacing keywords. Thinking about that solution showed us that it is bad practice because replacing certain parts of the query string only based on detecting keywords like "delete node" in this case and replacing it with "insert node [..]" is not a clean solution. If an element that has to be inserted contained one of these keywords (like "delete") in its name, this keyword would be replaced as well. This would alter the element that is to be inserted and thus put the database into an unwanted state.

Another approach that came to our mind is the usage of triggers that are supported e.g. by the DBMS Sedna [1]. The idea behind this concept was to fire a trigger every time an update is executed. So besides the update being processed, the trigger invokes queries that ensure the versioning of the dataset. This approach was turned down because of the fact that hardly any native XML database management system supports triggers. Thus it contradicted our goal to develop an approach, which fits multiple DBMS. In this thesis we focus on the two DBMS "BaseX" and "eXist-db" but with customizing other database management systems could be supported as well.

The next step in our process of thinking led us to the core of the DBMS "BaseX" itself. We wanted to implement a kind of a trigger system ourselves. Every time a CRUD operation, so to say a core operation, is invoked additional queries ensuring the versioning of the dataset are executed. Due to the fact that this happens directly inside the source code of the DBMS, nothing would change for the user. The advantage of this approach is that each query is parsed and thus checked for errors. The user receives notifications and error messages according to the inserted queries and additional queries are executed in the background without the knowledge of the user. The fact that every database management system would have had to be edited made this approach not universally useable. Furthermore not all the DBMS are open source which means that the source code of them cannot be edited.

[1] Compare `http://www.sedna.org/` - Last access: 14-09-2014

5.1.2 Turned Down Designs for Approaches for Versioning/Timestamping

Creation of Tracking Databases

The idea behind this approach, which if similar to "diff" algorithms, was the creation of two "tracking databases". The actual database (we will call it "productive database") is always up to date and does not contain any disabled nodes. These disabled nodes are stored in the two external tracking databases.

Every element node in the productive database contains two attributes: a unique ID that helps identifying that node and an attribute like "xmlDB_inserted", which shows the time this node was inserted on. When a node is deleted, it is moved to the first tracking database. Thus the first tracking database contains all the deleted nodes and additionally the ID of their parent nodes plus the information when they got deleted.

A second tracking database containing all the updated attributes exists as well. This database contains the following information: the ID of the node that contained this attribute, the name of the updated/deleted attribute, the old value and the time this attribute was updated/deleted at.

With the help of these two external databases, every former state of the productive database can be reconstructed. When the database is updated on a regular basis, restoring the original state of the database within a reasonable time becomes harder and harder. Furthermore both tracking databases have to be iterated through every time a query is re-executed. With each step of the iteration, the according node in the productive database has to be found. Due to the sheer amount of such looking up processes this approach got discarded. Additionally this approach requires an editing of the parser and thus the source code of the respective DBMS.

Creation of a Tracking Table II

Another version of the just described approach was not to assign a unique ID to every node but only to those that are updated. When a child of a certain node is deleted, an attribute like "childEdited" with a unique ID as value is inserted. If an attribute is edited or deleted, an attribute like "attributeEdited" is inserted. With that concept it is possible to look up old values while parsing the XML document. But like the previously presented approach, this one requires the editing of the respective database management systems XML Parser.

Single Node Selection with the Help of a GUI

This approach was about providing a GUI, where nodes that have to be edited are simply selected by the user via an ID. In this approach every element node has got an attribute with the name "citationID" that makes unique identifying of each node possible. This approach is pretty practical for the user but only allows him to edit a single node by entering the "citationID" for an element, the ID and the attribute name for an attribute and the ID plus the information that its text has to be edited, for a text node. What made us turning down this approach is the fact that this design restricts the user very heavily.

5.2 Approach I: Branch Copy

In this section the first approach we present in order to overcome the problem of the citation of subsets of XML data will be described. Every element node has got two additional attributes showing their validity, namely “xmlDB_inserted” and “xmlDB_deleted”. Thus these two attributes serve the purpose of showing the period of time they were valid in. If the “xmlDB_deleted” attribute is set to the value “null”, which is the default value, the node does not have an end of validity and thus has not been disabled yet.

In order to make citation possible, nodes must not be deleted or edited but only marked to show that changes happened. If e.g. an element node has to be deleted, it is just disabled and if it has to be edited, the element is copied - the old node is disabled and the new one edited accordingly. This means that not only the elements themselves but also all the children and thus whole branches are copied. This approach is an implementation of the approach presented by the authors of [70], which was described in Section 2.2.2. The difference between these two approaches is the naming of the attributes showing the validity of elements because we wanted to introduce another naming convention.

To get an idea of the functioning of this approach, an example for the “rename” operation is given at this point. Figure 5.1 shows the two steps needed to perform the modified update “rename” when a node of type element is addressed. The two boxes in the figure represent queries that have to be sent to the database. The list in each of the boxes defines the operations, which are invoked by the respective query.

With the first query, the identified element (target of the “rename” operation) is copied and marked with the help of a marking attribute. Due to the nature of XML both updates are performed at the same time. It is thus ensured that the newly copied node is not marked as well.

With the help of the second query, the old element node, which can easily be identified with the help of the marking attribute that was inserted in query one, has its marking attribute deleted. Furthermore the old element node and all its children of type element are disabled by editing the value of the attribute “xmlDB_deleted”. The new element node is updated accordingly - in this case renamed - and its “xmlDB_inserted” timestamp is edited and renewed. As a last step all the children of the new node, which are of the type element and already disabled, are deleted as the whole node was copied not minding whether children are active or not.

As a result after the execution of both queries, the old element is disabled and thus was valid from the time of its creation until the time of the update. The new node is edited accordingly (renamed in this example) and is valid from the time of the update without an expiration date (value of attribute “xmlDB_deleted” is “null”).

Figure 5.2 illustrates the rename example. The box named “node” represents an element node. The two boxes on the outer right are the same as the ones in Figure 5.1 and show the

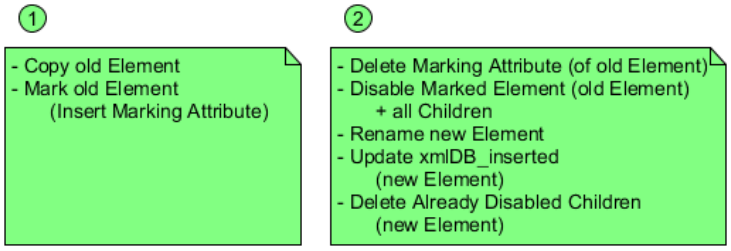

Figure 5.1: Approach I - "Branch Copy": Example for "Rename" Operation - Sequence of Events

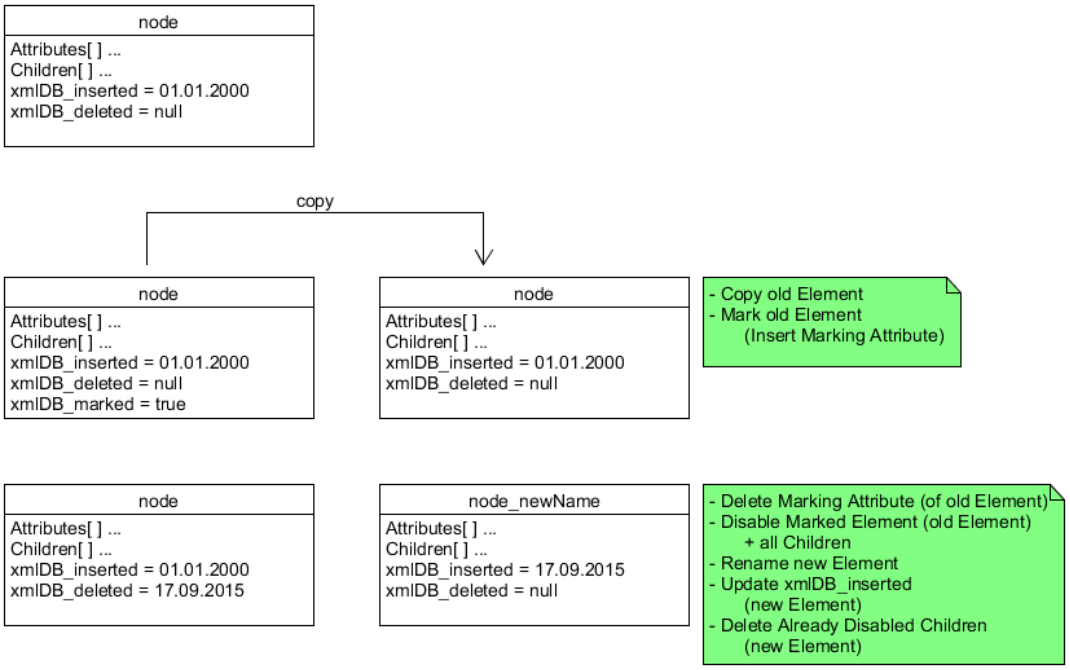

Figure 5.2: Approach I - "Branch Copy": Example for "Rename" Operation

changes that have been made to the node in the previous row in order to achieve the state of the database represented by the two other boxes in the respective line.

When CRUD operations are performed on the history dataset, only the valid nodes are taken into account. This means that the attribute timestamp contained by "xmlDB_inserted" has to be before (in the sense of time) the current timestamp and the attribute "xmlDB_deleted" has to be "null".

The following sections will describe all the five types of CRUD operations in detail and thus define this approach. Before reading these sections, it is important that one is aware of the fact that there are mainly two cases of modifying nodes. Targeting an element node requires slightly other actions than targeting an attribute or a text node.

5.2.1 Insert

The “insert” operation is different from the just presented example of the “rename” operation. There are two main cases identified by us, namely the insertion of an element node and the insertion of an attribute / a text node.

When an element has to be inserted, it is inserted with the help of the first query. Query two identifies all the nodes of the same type, selects the one that has just been inserted (possible due to the fact that the attributes “xmlDB_inserted” and “xmlDB_deleted” are missing), and inserts the two attributes “xmlDB_inserted” and “xmlDB_deleted”. All the element nodes on the “descendant”-axis are having these two attributes inserted as well.

When an attribute or a text node has to be inserted, the procedure is similar to the one described in Figure 5.1. The target element node is copied and marked with the first query. With the help of query two, the mark of the old target element is deleted, the old target element is disabled (including all the element nodes on the “descendant”-axis), the new attribute or text node is inserted into the copied and thus newly created element, the “xmlDB_inserted” timestamps of the new element itself and all the children on the “descendant”-axis are set to the time of the execution of the query and as a last action all the already disabled elements on the “descendant”-axis of the new element node are deleted.

5.2.2 Delete

When an element node has to be deleted, it has to be disabled by setting the “xmlDB_deleted” attribute to the time of the execution of the query. Of course all the element nodes on the “descendant”-axis of this element have to be disabled as well.

When an attribute or a text node has to be deleted, the procedure is again similar to the one described in Figure 5.1 but the target of these operations is the parent node (an element) of the attribute or text node which has to be deleted. Thus the parent node is copied and marked with the first query. With query two, the mark of the old parent node is deleted, the old parent node is disabled (including all the element nodes on the “descendant”-axis), the attribute or text node of the newly created parent node is deleted, the “xmlDB_inserted” timestamps of the new node itself and all the elements on its “descendant”-axis are set to the time of the execution of the query and as a last action all the already disabled elements on the “descendant”-axis of the new parent node are deleted.

5.2.3 Replace

The “replace” operation is a combination of the “delete” and the “insert” operation. When replacing an element with another element node, the old element and all the elements on its “descendant”-axis are disabled by setting the “xmlDB_deleted” timestamp and the new element node is inserted into the parent element of the target node with the help of the first query. The second query inserts the two required attributes “xmlDB_inserted” and “xmlDB_deleted” into the newly inserted element and all the elements on its “descendant”-axis.

When replacing an attribute with another attribute node the procedure is similar to the one described in Figure 5.1. With the first query the parent element node of the target attribute is copied and marked with the marking attribute. With the second query the mark of the old parent element node is deleted and additionally it is disabled (including all the elements on the “descendant”-axis). Following, the old attribute in the copied and thus newly created parent node is deleted and the new attribute is inserted. Furthermore the “xmlDB_inserted” timestamps of the new parent node itself and all the elements on its “descendant”-axis are set to the time of the execution of the query. As a last action all the already disabled elements on the “descendant”-axis of the new parent node are deleted.

When it comes to replacing a text node with a text, the same steps as for the operation of replacing an attribute with another attribute are taken.

5.2.4 Replace Value

The concept of the actions executed when a “replace value” operation is invoked is similar to the one of the “replace” operation.

When the value of an element node has to be replaced with a text, the target element is copied and marked with a marking attribute with the help of the first query. Query two deletes the mark of the old target element and disables it (including all the element nodes on the “descendant”-axis) by updating the “xmlDB_deleted” attributes. Furthermore all the children of the newly created element are deleted and the new text is inserted. This results in the text node being the only child of the newly created element node. Additionally the “xmlDB_inserted” timestamp of the new element is set to the time of the execution of the query.

When the value of an attribute or a text node has to be replaced, the procedure is exactly the one of the “replace” operation, which has already been described in Section 5.2.3. The only difference is that not the whole attribute but only the value of the attribute is replaced.

5.2.5 Rename

Renaming an element has already been explained with the help of the two figures 5.1 and 5.2. The only difference between renaming an element and an attribute is the target node. When an element node has to be renamed, the node itself is targeted by the queries sent to the DBMS by the parser. This means that the target element node is copied, the old one is disabled and the old one is renamed accordingly. When an attribute or a text node has to be renamed, the parent element node is the target of the operations. The possibility of renaming a text node is ignored because both of the database management systems used in this thesis yield errors when trying to do so.

5.3 Approach II: Parent-Child

Like Approach I - "Branch Copy", Approach II - "Parent-Child" uses the timestamps "xmlDB_inserted" and "xmlDB_deleted" to ensure versioning of element nodes. The big difference is that only the root node holds an "xmlDB_inserted" attribute from the beginning indicating when the whole dataset was created. Furthermore no "xmlDB_deleted" attribute exists in the whole dataset at the time of its creation. When an element node with several hierarchical levels is inserted, only the top level node receives an "xmlDB_inserted" timestamp showing when it was inserted. If an element node has to be deleted, a "xmlDB_deleted" attribute is inserted only into the target node showing that it was deleted. This means that not every element node holds the information when it was inserted or deleted - these pieces of information can be derived from their ancestor-axes [2].

Our second approach named "Parent-Child" is based on the "full implementation" version of the in Section 2.2.1 described approach. The big difference between the two approaches is that the authors of [4] rework the structure of the XML document completely as attribute and text nodes are stored in elements named "attribute" and "stringvalue". The approach presented by us keeps the original structure and adds additional elements that serve the purpose of versioning.

Because of the fact that this second approach is complex, we will provide an example story. The starting point of this story is the small dataset provided in Listing 5.1. Newly added or edited parts of the dataset will be highlighted. For this example story we simplified the timestamps so that they contain only the date and not the exact time - when using this approach proper timestamps are inserted.

Listing 5.1: Example Story - Dataset

```
<tmSegments xmlDB_inserted='01.01.2014'>
  <tmSegment att1='value1' att2='value2' att3='value3'>
    <segLabel>A</segLabel>
```

[2] compare Figure 3.1 on page 30

```
    <beginIndex>55</beginIndex>
    <endIndex>86</endIndex>
  </tmSegment>
</tmSegments>
```

5.3.1 Insert

In the first step depicted in Listing 5.2 a second “tmSegment” element node is inserted. It can be seen that only the top level element receives an “xmlDB_inserted” timestamp implying when it was inserted.

Listing 5.2: Example Story - Insertion of Element

```
<tmSegments xmlDB_inserted='01.01.2014'>
  <tmSegment att1='value1' att2='value2' att3='value3'>
    <segLabel>A</segLabel>
    <beginIndex>55</beginIndex>
    <endIndex>86</endIndex>
  </tmSegment>
  <tmSegment xmlDB_inserted='02.02.2014'>
    <segLabel>B</segLabel>
    <beginIndex>975</beginIndex>
    <endIndex/>
  </tmSegment>
</tmSegments>
```

Attribute and text nodes are versioned with the help of versioning blocks (with the name “xmlDB_version”), which are children of the element nodes the respective attributes or texts belong to. In Listing 5.3 an attribute as well as a text node are inserted. Because of the fact that a “xmlDB_version” element exists it can be seen that the attribute / text node either did not exist from the time of the creation of its parent element node on or was edited/deleted in the meantime. The attribute “type” in the “xmlDB_version” element indicates what type of node is versioned - the two possible types are “attribute” and “text”. When an attribute is versioned, the attribute “name” is needed in order to give information about the name of the attribute as there might be multiple ones in a single element node. The attribute “lastEdited” shows when the attribute was last edited. The last attribute in these versioning blocks is “nrPositions”, which will be described later on.

Listing 5.3: Example Story - Insertion of Attribute & Text

```
<tmSegments xmlDB_inserted='01.01.2014'>
  <tmSegment newAttribute='new Information' att1='value1' att2=
     'value2' att3='value3'>
    <segLabel>A</segLabel>
```

```
    <beginIndex>55</beginIndex>
    <endIndex>86</endIndex>
    <xmlDB_version type='attribute' name='newAttribute'
    lastEdited='03.03.2014' nrPositions='0'/>
  </tmSegment>
  <tmSegment xmlDB_inserted='02.02.2014'>
    <segLabel>B</segLabel>
    <beginIndex>975</beginIndex>
    <endIndex> 4823
      <xmlDB_version type='text' lastEdited='03.03.2014'/>
    </endIndex>
  </tmSegment>
</tmSegments>
```

5.3.2 Delete

When an element node is deleted, an attribute named "xmlDB_deleted" is inserted into the element. Listing 5.4 deletes the just inserted "tmSegment" element again.

Listing 5.4: Example Story - Deletion of Element

```
<tmSegments xmlDB_inserted='01.01.2014'>
  <tmSegment newAttribute='new Information' att1='value1' att2='
     value2' att3='value3'>
    <segLabel>A</segLabel>
    <beginIndex>55</beginIndex>
    <endIndex>86</endIndex>
    <xmlDB_version type='attribute' name='newAttribute'
       lastEdited='03.03.2014' nrPositions='0'/>
  </tmSegment>
  <tmSegment xmlDB_deleted='04.04.2014' xmlDB_inserted='
     02.02.2014'>
    <segLabel>B</segLabel>
    <beginIndex>975</beginIndex>
    <endIndex>4823
      <xmlDB_version type='text' lastEdited='03.03.2014'/>
    </endIndex>
  </tmSegment>
</tmSegments>
```

The next step depicted in Listing 5.6 shows the deletion of a text as well as an attribute node. The text node that has to be deleted belongs to the "segLabel" element of the first "tmSegment" element node. Due to the fact that under this path no text has ever been inserted, edited or deleted since the dataset was created, a completely new versioning block ("xmlDB_version")

has to be inserted. The children named “xmlDB_v” show the different versions and thus the history of the text node. It can be seen that the text node held the value “A” from “01.01.2014” till “05.05.2014”. The attribute “lastEdited” is set to “deleted” as it is not valid anymore.

When it comes to attributes the versioning is a bit more complicated as the ordering of attributes is important. XML files with the same content but different attribute orderings have got different md5 hashes and thus it has to be ensured that the original position of a deleted attribute has to be maintained. For this purpose we use a concept of so called “spacing attributes” (term defined by us). If an attribute is deleted it is simply replaced by such a spacing attribute in order to “reserve” this spot. When a select query is re-executed, the original attribute can take this spot again. Listing 5.5 shows the naming convention of such spacing attributes.

Listing 5.5: Naming Convention of Spacing Attributes

```
xmlDB_[name of attribute]_position[position number]
```

Due to the possibility of deleting an attribute and inserting an attribute with the same name again, multiple positions and thus spacing attributes have to be tracked for a single attribute (an attribute is always inserted as first into the list of attributes an element holds). The information of how many positions are currently tracked for a respective attribute is stored in the attribute “nrPositions” in the “xmlDB_version” element. The “[position number]” part of the naming depicted in Listing 5.5 represents this number making unique identification of spacing attributes possible. In our example depicted in Listing 5.6 the attribute “att2” in the first “tmSegment” element has been deleted one time - this means that the “nrPositions” attribute is set to “1” because one spacing attribute is active. The spacing attribute is called “xmlDB_att2_position1” in our case. The reference between the different versions of attributes and their original positions is established with the help of the “originalPosition” attribute within the respective “xmlDB_v” element node. A last information to be given at this point is that not all the “xmLDB_v” elements hold such an “originalPosition” attribute as it is only inserted when the attribute is deleted (the position would get lost otherwise).

Listing 5.6: Example Story - Deletion of Attribute & Text

```
<tmSegments xmlDB_inserted='01.01.2014'>
  <tmSegment newAttribute='new Information' att1='value1'
     xmlDB_att2_position1='t' att3='value3'>
    <segLabel>
      <xmlDB_version type='text' lastEdited='deleted'>
        <xmlDB_v xmlDB_inserted='01.01.2014'
        xmlDB_deleted='05.05.2014'>A</xmlDB_v>
      </xmlDB_version>
    </segLabel>
    <beginIndex>55</beginIndex>
    <endIndex>86</endIndex>
```

```
    <xmlDB_version type='attribute' name='newAttribute'
       lastEdited='03.03.2014' nrPositions='0'/>
    <xmlDB_version type='attribute' name='att2'
    lastEdited='deleted' nrPositions='1'>
      <xmlDB_v xmlDB_inserted='01.01.2014'
      xmlDB_deleted='05.05.2014'
      originalPosition='xmlDB_att2_position1'>value2</xmlDB_v>
    </xmlDB_version>
  </tmSegment>
  <tmSegment xmlDB_deleted='04.04.2014' xmlDB_inserted='
     02.02.2014'>
    <segLabel>B</segLabel>
    <beginIndex>975</beginIndex>
    <endIndex>4823
      <xmlDB_version type='text' lastEdited='03.03.2014'/>
    </endIndex>
  </tmSegment>
</tmSegments>
```

5.3.3 Replace

The replacement of an element with another element is a combination of an insertion and a deletion of an element and is thus not described again. Replacing an attribute requires to differentiate between to cases - in the first one the new attribute is named exactly like the old one and in the second one the names are different. If the names are equal the "replace" operation is processed like a "replace value" operation which means that only the value of the attribute is updated. If the names are equal the old attribute is deleted and the new one is inserted, which means that it is a combination of the "insert" and "delete" operation. The replacement of text nodes is again a combination of "insert" and "delete".

5.3.4 Replace Value

The combinations "replace value ELEMENT with ELEMENT" and "replace value TEXT with TEXT" are again simply "insert" and "delete" operations. A more interesting case is the replacement of the value of an attribute with a text. To show this behaviour Listing 5.7 depicts the replacement of the value of the attribute "newAttribute". Due to the fact that a versioning block already exists, only a new "xmlDB_v" element that contains the former state has to be created.

Listing 5.7: Example Story - Replace Value of Attribute

```
<tmSegments xmlDB_inserted='01.01.2014'>
  <tmSegment newAttribute='reworked Information' att1='value1'
     xmlDB_att2_position1='t' att3='value3'>
    <segLabel>
```

```
    <xmlDB_version type='text' lastEdited='deleted'>
      <xmlDB_v xmlDB_inserted='01.01.2014' xmlDB_deleted='
         05.05.2014'>A</xmlDB_v>
    </xmlDB_version>
  </segLabel>
  <beginIndex>55</beginIndex>
  <endIndex>86</endIndex>
  <xmlDB_version type='attribute' name='newAttribute'
     lastEdited='06.06.2014' nrPositions='0'/>
    <xmlDB_v xmlDB_inserted='03.03.2014'
    xmlDB_deleted='06.06.2014'>new Information</xmlDB_v>
  </xmlDB_version>
  <xmlDB_version type='attribute' name='att2' lastEdited='
     deleted' nrPositions='1'>
    <xmlDB_v xmlDB_inserted='01.01.2014' xmlDB_deleted='
       05.05.2014' originalPosition='xmlDB_att2_position1'>
       value2</xmlDB_v>
  </xmlDB_version>
 </tmSegment>
 <tmSegment xmlDB_deleted='04.04.2014' xmlDB_inserted='
    02.02.2014'>
  <segLabel>B</segLabel>
  <beginIndex>975</beginIndex>
  <endIndex>4823
    <xmlDB_version type='text' lastEdited='03.03.2014'/>
  </endIndex>
 </tmSegment>
</tmSegments>
```

5.3.5 Rename

For the "rename" operation we differentiate two cases, namely renaming an element and renaming an attribute. Element nodes that have to be renamed are versioned with the help of versioning blocks again. Listing 5.8 depicts the case of renaming an element. In this example the first "tmSegment" element is renamed as "newElementName". The value of the attribute "type" in the newly created versioning block indicates that the parent node was renamed and archives its old name, which was valid from "01.01.2014" till "07.07.2014" (compare the timestamps "xmlDB_inserted" and "xmlDB_deleted" in the "xmlDB_v" element).

The "rename" operation for attributes is performed as a combination of the "delete" and the "insert" operation. This is important to mention because of the fact that besides the name of the respective attribute also its position changes. We implemented this approach in this way because

of lower complexity and performance reasons when queries are to be re-executed and subsets have to be exported.

Listing 5.8: Example Story - Renaming of Element

```
<tmSegments xmlDB_inserted='01.01.2014'>
  <newElementName newAttribute='reworked Information' att1='
     value1' xmlDB_att2_position1='t' att3='value3'>
    <segLabel>
      <xmlDB_version type='text' lastEdited='deleted'>
        <xmlDB_v xmlDB_inserted='01.01.2014' xmlDB_deleted='
           05.05.2014'>A</xmlDB_v>
      </xmlDB_version>
    </segLabel>
    <beginIndex>55</beginIndex>
    <endIndex>86</endIndex>
    <xmlDB_version type='attribute' name='newAttribute'
       lastEdited='03.03.2014' nrPositions='0'/>
      <xmlDB_v xmlDB_inserted='03.03.2014' xmlDB_deleted='
         06.06.2014'>new Information</xmlDB_v>
    </xmlDB_version>
    <xmlDB_version type='attribute' name='att2' lastEdited='
       deleted' nrPositions='1'>
      <xmlDB_v xmlDB_inserted='01.01.2014' xmlDB_deleted='
         05.05.2014' originalPosition='xmlDB_att2_position1'>
         value2</xmlDB_v>
    </xmlDB_version>
    <xmlDB_version type='renameE' lastEdited='07.07.2014'>
      <xmlDB_v xmlDB_inserted='01.01.2014'
      xmlDB_deleted='07.07.2014'>tmSegment</xmlDB_v>
    </xmlDB_version>
  </newElementName>
  <tmSegment xmlDB_deleted='04.04.2014' xmlDB_inserted='
     02.02.2014'>
    <segLabel>B</segLabel>
    <beginIndex>975</beginIndex>
    <endIndex>4823
      <xmlDB_version type='text' lastEdited='03.03.2014'/>
    </endIndex>
  </tmSegment>
</tmSegments>
```

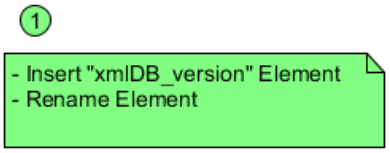

Figure 5.3: Approach II - “Parent-Child”: Sequence of Events

5.3.6 Short Summary & Example

After the description of all the five CRUD operations, this section will provide a summary and additional information concerning the functioning of this approach. The different cases for the versioning of text/attribute nodes are listed subsequently.

- **No text/attribute node & no “xmlDB_version” element exists:** A text/attribute node has never existed.
- **Text/Attribute node exists & no “xmlDB_version” element exists:** The text/attribute node has existed since its parent element node was inserted.
- **Text/Attribute node exists & “xmlDB_version” element exists:**
  - No “xmlDB_v” element exists: In this case a text/attribute node was inserted into an element where no text/attribute node (with this name in the case of an attribute node) has ever existed.
  - One to multiple “xmlDB_v” elements exist: In this case multiple versions of text/attribute nodes (with the same name in the case of an attribute node) exist for the respective element node. The timestamps “xmlDB_inserted” and “xmlDB_deleted” in each of the “xmlDB_v” elements indicate which version was valid at which point in time. The current version of the text/attribute node is never stored within the “xmlDB_version” block because it would be redundant to do so.
- **No text/attribute node exists & “xmlDB_version” element exists:** The text/attribute node was deleted in the past. This information can also be derived from the attribute “lastEdited” in the “xmlDB_version” element as the value is “deleted” if the text/attribute node does not exist any longer.

In order to be able to compare the two approaches we will use the example given in the context of Approach I - “Branch Copy” in Section 5.2 again. Figure 5.3 shows the steps which are needed in order to perform the modified update “rename” when a node of type element is addressed. The single box indicates that only one query is sent to the DBMS and the content of it represents the operations that are invoked by this query.

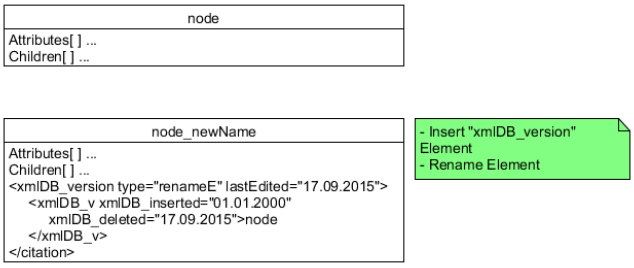

Figure 5.4: Approach II - "Parent-Child": Example for "Rename" Operation

Figure 5.4 shows again which steps are to be taken and how the element node is edited. The box with the name "node" represents the element node that is to be renamed. The green box on the right side is the same one as the one in Figure 5.3 and shows the changes which were invoked in order to achieve the new state of the element node.

5.4 Approach Independent Operations

In this section we will briefly address three operations that are independent of the two approaches presented in the Sections 5.2 and 5.3.

GenerateSubset

Every query that neither contains a CRUD nor an "executeAndArchive" or a "re-execute" operation (described subsequently) only derives data from the database and is thus a simple "select" statement. When such an operation is invoked, a subset is generated from the up to date database and provided to the user. Users are given the possibility to display small subsets (max. 50MB) directly in the GUI or download them. Bigger datasets can only be downloaded with the help of a download link.

ExecuteAndArchive

The "executeAndArchive" operation lets users archive "select" queries that do not invoke updates on the database and thus only derive a subset of the database. Furthermore the subset identified by the query is offered as a download after having archived it. Providing this subset is done exactly like in the "generateSubset" operation.

Archived queries are stored in the query store, which is described in Section 5.6.2. Besides the assigned PID and the query itself metadata like the size of the XML file, its md5 hash and queries that are needed for the re-execution are stored in the query store.

When an “executeAndArchive” operation is called, the md5 hash of the select query entered by the user is calculated as a first step. Secondly, the subset identified by the select query is appended and its md5 hash is computed as well. If both md5 hashes match an already existing entry in the query store and thus a query identifying the exact same subset already exists, the PID of the already stored query is displayed to the user. If the subset is new, the chosen PID provider generates a PID and afterwards the query including all the needed data is stored in the query store. As a last step the generated XML file is sent to the user. The general structure of this operation is as follows: “executeandarchive [QUERY]”.

Re-execute

With the help of the “re-execute” operation users are given the possibility to re-execute the queries archived in the query store. The history database is copied and edited so that its state is the same as at the time the query was executed. How this is done depends on the respective approach the database is versioned with. Afterwards a subset is generated by invoking the stored query. Like for the “re-execute” operation, subsets with a maximum size of 50MB can be displayed directly in the GUI.

Concerning the copying of the history database we also considered directly exporting a subset with the help of e.g. XLST but the transformations turned out to perform badly when it comes to subsets bigger than 10 MB (around one minute of execution time for 10 MB). Furthermore not fully copying the history database but creating a new one containing only the identified subset is a possibility. In this case the transformations that are needed in order to restore earlier states of the subset have to be executed only on this subset. A problem with this solution is that copying a whole database performs way better than creating a new one. A combination of the ideas of copying the whole database and creating a new one containing only the identified subset is possible. In this case the chosen approach would depend on the size of the identified subset - if the subset covers most parts of the database, the whole one would be copied. If the subset is small compared to the overall size of the history database, the latter one would be chosen. However, this combination is not covered in this thesis and thus is an interesting topic for future work.

The general structure of this operation is as follows: “reexecute [PID]”.

5.5 Implementing the Versioning/Timestamping

The findings that the authors of [54] made for relational database management systems are also valid for native XML database management systems. In the context of RDBMS “update”, “insert” and “delete” statements have to be tracked in order to be able to document all the changes made to the dataset and to be able to reproduce a specific state of a database or a subset of it at a certain point in time. For a native XML DBMS these CRUD operations are “insert”, “delete”, “replace” and “rename”. The fifth possible operation, “replace value” is seen as a subtype of the “replace” operation.

In order to make XML databases citable we identified the following prerequisites and made the assumptions listed below. To provide a better overview we divided this section into general prerequisites (Section 5.5.1) and approach specific prerequisites (Section 5.5.2).

5.5.1 General Prerequisites

1. One important assumption we made is that people using this parser know how XML databases are used and thus how XQuery and XPath work. Furthermore users are assumed to know the dialect of the database management system "BaseX". This means that users are capable of writing queries that fit the required structure and dialect of "BaseX" (only queries with this dialect are invoked). Which DBMS operates in the backend does not matter because of the fact that the parser identifies the needed dialect and syntax and creates proper queries for the respective DBMS.

2. An assumption which is closely related to the first one is that users do not try to insert an attribute node if an attribute with the same name already exists in the target element node.

3. The IP of the server where the parser, which is presented in this thesis, is located has to be known at any point in time. Furthermore the name of the dataset the user wants to query has to be known as well. Information on the IP of the database management system or its type (e.g. "BaseX" or "eXist-db") is read from a configuration file at the start of the server of the parser. The configuration file will be described in detail in Section 5.6.4.

4. Another assumption or prerequisite we identified is that in every dataset there exists only one active root node (addressed by "/*") at any point in time. Active means that the attribute "xmlDB_inserted" is smaller and the attribute "xmlDB_deleted" (if existing when it comes to Approach II) is greater than the timestamp of the time the query is executed at. This assumption is a general one as XML datasets are defined to have exactly one root node (compare Section 3.1).

5. Each dataset has got a query store and a history database that are accessible under the same IP as the dataset itself. The name of the query store is the name of the dataset with an added "_Query_Store". So the query store of the dataset "measurement_db" would be "measurement_db_Query_Store" for example (compare Section 5.6.2). The history database of each dataset is named exactly like the dataset itself with an added "_History" string. Thus for example the history database of the dataset "measurement_db" would be "measurement_db_History".

6. It is assumed that element nodes only contain either elements or a single text node as children/child. Mixing element and text nodes is excluded. This means that when text nodes are addressed with the help of the expressions "/child::text()" or "/text()", one can be sure that exactly one text node is addressed per parent element node. Furthermore users are assumed not to invoke CRUD operations that mix up these two types of nodes.

7. User defined variables (defined with the following syntax: $[variableName]) must not start with the string "xmlDB_" because of the fact that such variables are reserved for the

parser. Same goes for any content the users wants to insert into the databases. This means that the names of elements and attributes must not start with the string "xmlDB_" as well.

8. The last assumption we made is that datasets in DBMS are accessible and editable for every user accessing them via our parser. To ensure this for the DBMS "BaseX", we use the "admin" account for every query sent to the DBMS by the parser. When it comes to "eXist-db", we define that the group "GUEST" is allowed to edit the respective dataset [3].

5.5.2 Approach Specific Prerequisite

1. Approach I - "Branch Copy": Every element node has to contain the two timestamps "xmlDB_inserted" and "xmlDB_deleted" in order to allow versioning.

5.6 Architecture

Figure 5.5 depicts the general approach we present in order to achieve the goal of making datasets citable. We created a parser based on the database management system "BaseX" that acts as a middleware between the user and the DBMS. Every query entered by the user is therefore parsed and rewritten to meet the specifications of the respective approach. After that, the edited query or queries is/are sent to the DBMS. How data are stored and how versioning is achieved depends on the specific approach.

The database management system in the backend presented in Figure 5.5 is replaceable due to the fact that the parser supports different types of "dialects" [4]. We depict the three databases "Scientific Data - up to date", "Scientific Data - History" and "Query Store" (described in Section 5.6.2) as separate ones because they are stored independently from each other.

For both our approaches we store scientific data in two different databases. One contains all the current records without archived data ("Scientific Data - up to date" in Figure 5.5) and the second one is a history database that keeps track of all the updates performed on the scientific data. This means that queries creating a subset of the latest version of the database target the up to date database and queries that re-execute archived queries target the history database. The advantage compared to having only the history database is that the newest state of the database does not have to be derived every time a subset is created.

The following sections will provide descriptions of basic components that will be used for both of the approaches and thus throughout this thesis. Section 5.6.1 will give an overview and will explain the concept of the query parser we implemented and the two Sections 5.6.2 and 5.6.3 will describe the concepts of a query store and a PID provider mockup web service. Both the query store and the mockup of a PID provider were implemented by us as well. A

[3] For an introduction on permissions for the DBMS "eXist-db" please check `http://exist-db.org/exist/apps/doc/security.xml` - Last access: 15-09-2015

[4] Every DBMS requires CRUD operations to have a specific and from system to system different syntax.

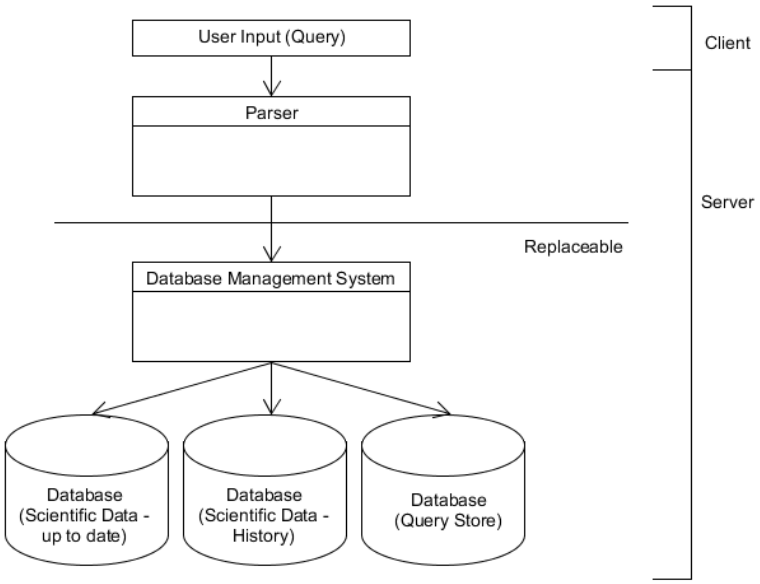

Figure 5.5: General Approach for Data Citation for Native XML Databases

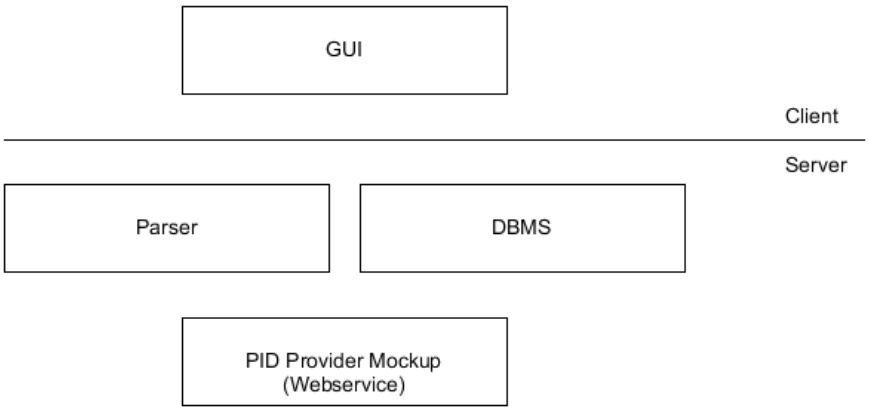

Figure 5.6: General Approach for Data Citation for Native XML Databases - Components

configuration file including all the pieces of information that are needed in order to run the parser will be described in Section 5.6.4. Furthermore the exception handling (5.6.5), additionally used external libraries (5.6.6) and the step by step processing of queries (5.6.7) will be described. Finally the graphical user interface will be depicted and described in Section 5.6.8.

5.6.1 Query Parser

The parser consists of five packages containing 20 classes in total. The package “org.basex” contains numerous packages and classes that are a part of the DBMS “BaseX” and needed for the parsing of the queries. We only edited the class “QueryParser” in the package “org.basex.query” as this class is the heart of the query parser of “BaseX”. This parser is based on version 7.7.2 of the DBMS.

The Figures 5.7 and 5.8 depict the structure of the newly added or edited Java packages and classes. Although it is one diagram, it is split into two separate graphics due to the limited space a page provides.

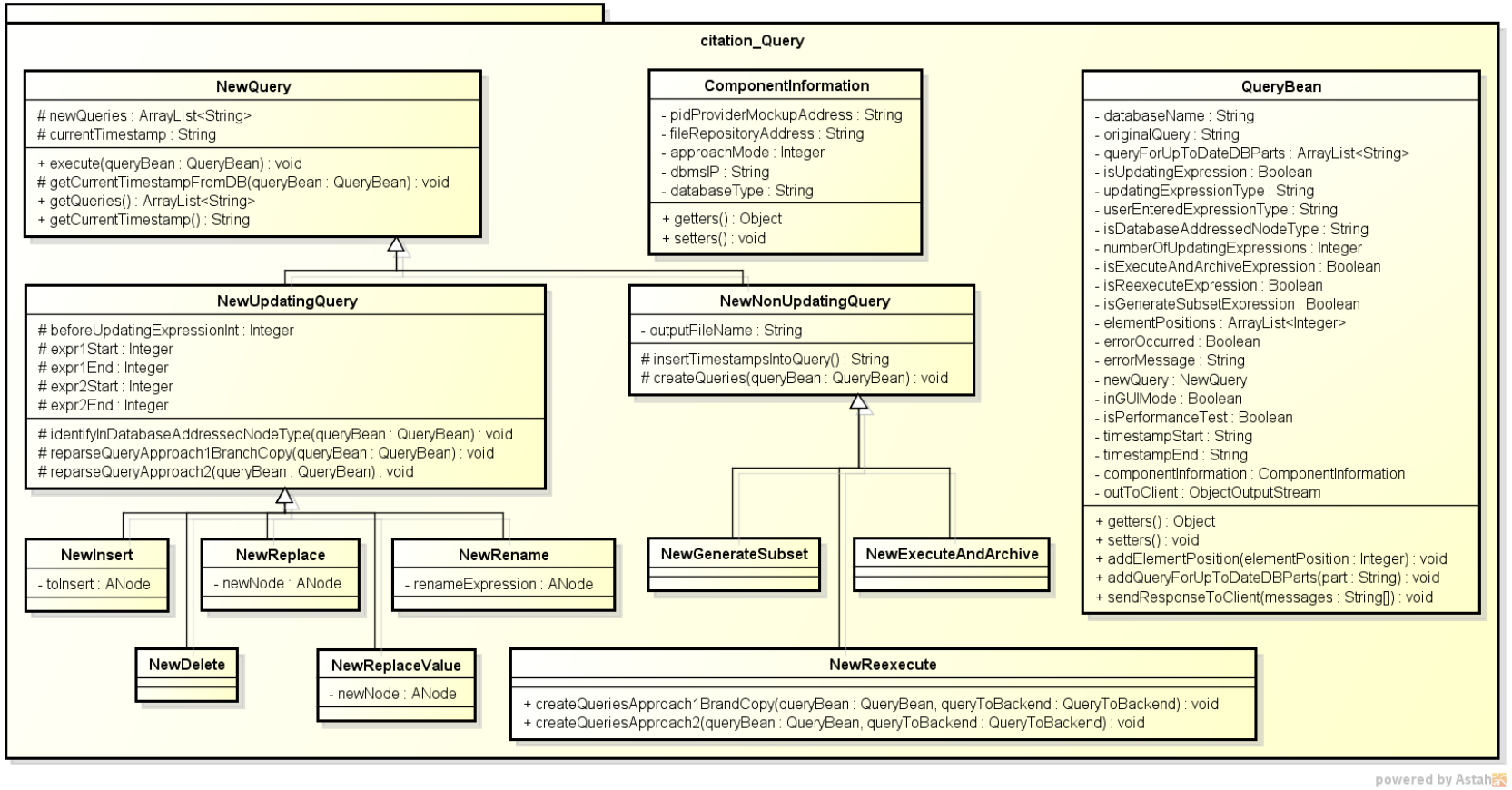

Figure 5.7: Class Diagram of Query Parser implemented by us - Part 1

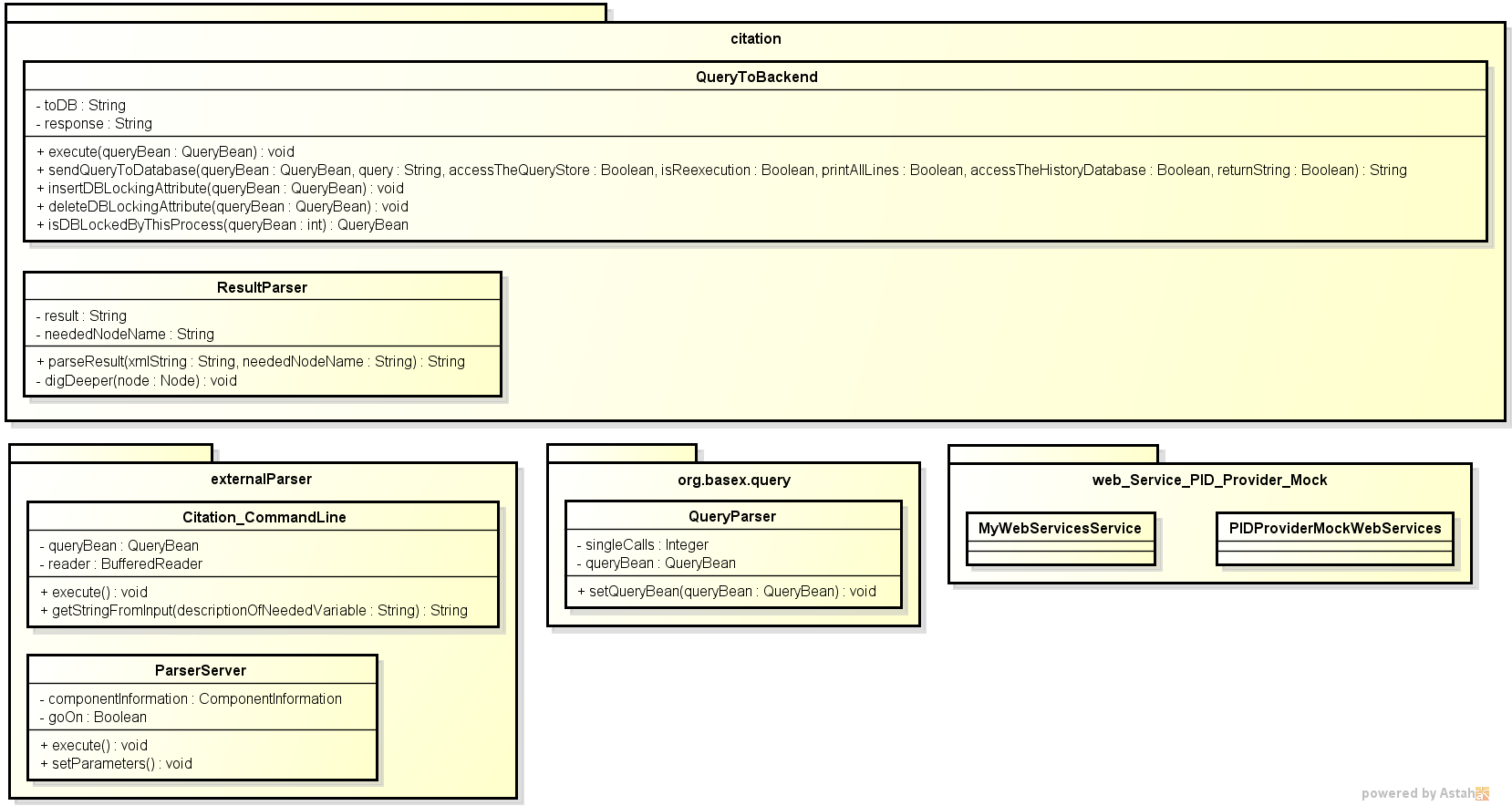

Figure 5.8: Class Diagram of Query Parser implemented by us - Part 2

For our approaches we implemented the subsequently listed consistency checks. We divided them into two categories - the ones that are checked by the parser presented by us (Section 5.6.1.1) and those that are not (Section 5.6.1.2). When the parser finds a violated check, an error is thrown. The consistency assumptions that are not checked will not throw an error message and thus users are assumed not to invoke these kinds of actions.

5.6.1.1 Consistency Checks done by the Parser

1. Only one CRUD operation is allowed per user input. This means that per query placed by the user, only one CRUD expression is allowed. This makes parsing and rewriting the queries possible. If the inserted query contains more than one CRUD expression an error is thrown and displayed to the user.

2. CRUD expressions are not allowed to edit different types of nodes in the course of one single call. So for example the user must not insert a certain node into an element node as well as into an attribute node at the same time. It is obvious that this does not find application in practice for the "insert" operation due to the fact that it is simply not possible to insert an element node into an attribute node. For the "delete" operation this would technically be possible but as targeting different types of nodes within a single query does not find application we did not include this functionality.

3. When it comes to the "insert" operation only one node - no matter what type - can be inserted per query. Thus it is not allowed to insert two nodes into a target element with a single query. We made this specification because of the fact that we focus on elemental updates in the first version of this implementation. It also applies for the operations "replace", "replace value" and "rename" as well. Multiple nodes of the same type can be selected as target nodes but it is only allowed to provide a single user entered node that should replace the target node, replace the value of the target node or rename the target node. Making the just defined operations possible is part of our future work.

4. In this version of the parser, the "insert" operation can only be invoked as "insert into". If nodes are to be inserted "after" / "before" a target node or "as first" / "as last" node, the additional information is ignored and the "insert" operation is executed as a normal "insert into" call. The implementation of these special cases is part of the future work.

5. When using the DBMS "BaseX" new nodes can only be inserted into a single element node. For the DBMS "eXist-db" new nodes are allowed to be inserted into not only a single but also multiple element nodes. When using the parser presented by us, we allow the user to insert nodes into multiple targets regardless what database management system is accessed in the end. This was made possible in order to standardise the "insert" operation a little bit more. The same applies for the operations "replace", "replace value" and "rename".

5.6.1.2 Consistency Assumptions not Checked by the Parser

1. The first assumption not being checked by the parser is a general one. In this thesis we put the focus on the three node types "element", "attribute" and "text" due to the fact that other node types like comments are only rarely used in practice. Furthermore the datasets we will use throughout this thesis only contain the three just mentioned node types.

2. The dataset must not contain any loops/cycles because we identified that it is not good practise to design a dataset that way.

   For better understanding, an example of a dataset containing loops is given subsequently in Figure 5.9. The example was constructed by us but is based on the XML file described in Figure 3.1 on page 24. In this dataset it can be seen that the element node "tmSegment" has another element named "tmSegment" as a child.

   Listing 5.9: Detecting Changes in XML Documents - XML File

```
<tmSegments>
  <tmSegment>
       ..
    <tmSegment>
      ..
    </tmSegment>
  </tmSegment>
</tmSegments>
```

3. If an element node is updated, no other node on its "descendant" or "ancestor" axis is updated in that query. With the already stated assumptions of having no loops in the dataset and not being able to update nodes of different types, we could not identify any practical application of such an operation and thus we assume that users do not invoke such operations.

4. The DBMS "BaseX" defines that all the entered expressions within one query must either only return values or be CRUD operations and return empty sequences. Thus it is not possible that nodes are edited and other nodes are displayed with the help of one single query. The DBMS "eXist-db" allows this combination but due to the fact that we want to keep the approach as generic as possible, we expect users not to combine the two types of queries.

5. Existing nodes in the database are only allowed to be used as target nodes. This means that e.g. the user is not allowed to insert an already existing element node (identified by an XPath expression) into another already existing element. Another example is the replacement of an element node by another element node that is already in the database. In our eyes these operations only find application if the schema of the database is completely modified. For these operations one has to have deeper insights into the database and its

working and thus should be capable of reworking the database accordingly without the use of this parser.

6. The last assumption not being checked by the parser is that element nodes are never identified with the help of their position in the dataset. Doing so would yield errors when using Approach I - "Branch Copy" because of the fact that this approach is based on copying element nodes. If elements were identified by their position, newly created (copied) element nodes could never be addressed. This means that the database would not be updated accordingly and a wrong state of the database would be the result. Making an identification based on the position of element nodes would have been possible at the cost of query performance but we identified that this case is not practicable and thus we decided against an implementation.

5.6.2 Query Store

The concept of a query store, which has already been mentioned in the context of relational databases in Section 2.1.1, is an important part of our solution as it stores queries that users want to archive in a database. Select queries, so to say queries that do not modify the dataset and just query information, can be stored in this database in order to make re-execution of experiments possible. As we have already identified as a prerequisite, every dataset has got a query store that is accessible under the same IP as the dataset itself. The query store is named exactly like the dataset with the string "_Query_Store" added at the end of the name.

Listing 5.10 depicts the structure of the query store. It can be seen that each "query"-element has got a couple of child nodes of type element that store information. The first of these nodes, namely "pid", stores the persistent identifier of the query. The "timestamp" node gives information on the time this query was executed and archived. The element node "uneditedQuery" stores the query in the way it was entered by the user and the element "queryMd5" stores its md5 hash. In the subsequently listed example, the user asked for all "measurement" elements. The nodes called "sizeOfXMLFileInBytes" and "fileMd5" contain information about the .xml file representing the respective subsets. The node "sizeOfXMLFileInBytes" represents the size of the .xml file in bytes and "fileMd5" represents the md5 hash of that file.

The last pieces of information being stored in the query store are the queries that are needed to re-execute the query and thus to reconstruct the generated subset. The element "queriesForReexecution" holds multiple "queryForReexecution" elements, which represent single queries. Their attributes named "id" represent the order in which these queries have to be executed.

Listing 5.10: Query Store

```
<archivedQueries>
  <query>
    <pid>23</pid>
    <timestamp>2014-10-01T12:00:00.000+02:00</timestamp>
    <uneditedQuery>//measurement</uneditedQuery>
```

```
    <queryMd5>cccfd9988cd5aecf3815c4ca1048b3e8</queryMd5>
    <sizeOfXMLFileInBytes>1273</sizeOfXMLFileInBytes>
    <fileMd5>5DF2E3A9C5CE598F63728F72AC2BC4C5</fileMd5>
    <queriesForReexecution>
      <queryForReexecution id="1">Query for Reexecution 1</
         queryForReexecution>
      ...
    </queriesForReexecution>
  </query>
  <query>
    ..
  </query>
  ..
</archivedQueries>
```

5.6.3 PID Provider Mockup Web Service

Persistent identifiers (PIDs) are needed in the context of the query store described in Section 5.6.2 as they identify the archived queries. Due to the fact that assigning real PIDs to data in a non productive environment is not practicable, we created a mockup of such a provider. This mockup is a web service that is run on the localhost and was created with the help of the tool wsimport [5].

The implementation of the client-side of the PID Provider Mockup Web Service can be found under the path "basex-core/src/main/java/web_Service_PID_Provider_Mock". The server-side is located in another project as it is independent from the parser presented by us. The two files needed for the web service can be found in the project "PID-Provider-Mock-Server" under the path "src/def.pkg".

To start the PID Provider Mockup Web Service on the localhost, the class "PublishWsOnServer" has to be run with the help of the method "start()" (automatically started by the "main(String[] args)" method of this class). To keep the implementation of this mockup simple, a pop-up with a button is displayed. By clicking this button the mockup is termined again.

The method "nextPid(...)" in the class "MyWebServices" accesses the query store of the respective dataset in the DBMS. The query sent to the DBMS queries the maximum number of all the PIDs, adds 1 to this value and returns it afterwards. The string returned by the DBMS is structured like the one depicted in Listing 5.11 while [PID] represents the value of the requested PID. With the help of this returned string the value of the needed PID is extracted and returned to the query parser in the form of a string containing only the integer value of the requested PID. Although the PIDs provided by this mockup are of type integer, a string is returned because

[5] Compare `http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html` - Last access: 15-07-2015

of the fact that replacing this mockup becomes easier as real PIDs are most likely not of type integer (compare Section 2.3).

Listing 5.11: Syntax of String Returned by the DBMS when the PID Provider Mockup Web Service Generates a PID

```
PID for this Query: [PID] PIDNumber
```

5.6.4 DBMS Configuration File

Information on the database management system (e.g. IP, its type, etc.) being addressed in the backend is stored in a java class named "ComponentInformation.java" under the path "basex-core/src/main/java/citation_Query". At the start of the server of the parser the user is asked to enter a path to a configuration file holding the required information about the DBMS. This information is read and stored in the just named java class. Listing 5.12 depicts the structure of the configuration file and thus reveals the pieces of information required.

Listing 5.12: Structure of Configuration File

```
<pidProviderMockupAddress>[Address of PID Provider Mockup
   Address]</pidProviderMockupAddress>
<fileRepositoryAddress>[Local Path of File Repository]</
   fileRepositoryAddress>
<approachMode>["1" or "2"]</approachMode>
<dbmsIP>[IP of Server]</dbmsIP>
<databaseType>["BaseX" or "eXist-db"]</databaseType>
```

5.6.5 Exception Handling

In this section we will target the topic of error and exception handling. Generally two types of errors can occur - syntactic and semantic errors.

Syntactic errors are related to the syntax of queries and thus occur when the query is not valid in terms of syntax. Due to the fact that our parser is based on the database management system "BaseX" and users only enter queries in the dialect of "BaseX" (identified as a prerequisite in Section 5.5.1), these errors are easy to handle as the parser checks the entered queries for mistakes automatically. In these cases the thrown errors are displayed to the user.

Semantic errors refer to problems that arise e.g. due to improper CRUD operations or the database being offline. For these errors we implemented the standard that the respective error message is surrounded with "<Error>...</Error>" tags and afterwards displayed to the user. As an example, Listing 5.13 depicts the error that is shown if the database management system is offline.

Listing 5.13: Error Displayed to the User When the DBMS is Offline

```
<Error>Server is offline! Please try again later.</Error>
```

In detail semantic errors occur in the three subsequently listed cases.

- The first case is when a CRUD operation that is not allowed for the respective DBMS type is invoked. We identified all the allowed CRUD operations in Section 4.2.3. An example for a CRUD operation that is not allowed is the insertion of an attribute node into an attribute node when using the DBMS "BaseX".

- The second case is when a problem with the database management system occurs. This means that errors are thrown when the DBMS is offline or currently locked by another process but also includes the case that a PID entered by the user for the re-execution of an archived query cannot be found in the query store of the database.

- The last case where a semantic error occurs is when consistency checks that we identified in Section 5.6.1.1 are violated. An example for such a case is the occurrence of multiple CRUD operations within a single query. In this case the error "<Error>Only one CRUD Operation per Query allowed.</Error>" would be thrown and displayed to the user.

5.6.6 Additionally used External Libraries

External libraries that are used by the parser are listed below.

- Apache HTTP Client (4.3.6)
  This library is used for sending the required queries to the database management system via HTTP. The HTTP Client is a component of the Apache HttpComponents project by the Apache Software Foundation [6].

- Apache Xerces (2.9.0) [7]
  If our parser is run in GUI mode, the SAXParser of this library is needed to convert the returned string into a SAXTree that is displayed in the tree window.

5.6.7 Processing of a Query

In this section we will cover the processing of queries by the parser presented in this chapter. This description will reach from the user invoking a query to receiving the output being returned by the DBMS. The bold heading of the respective sections represents the name of the class actions are executed in. Figure 5.9 depicts an ordered but rough (syntactically not completely correct) structure of the components that are invoked in the processing of queries.

[6] Compare `http://www.apache.org/` - Last access: 11-02-2015
[7] Compare `http://xerces.apache.org/` - Last access: 11-02-2015

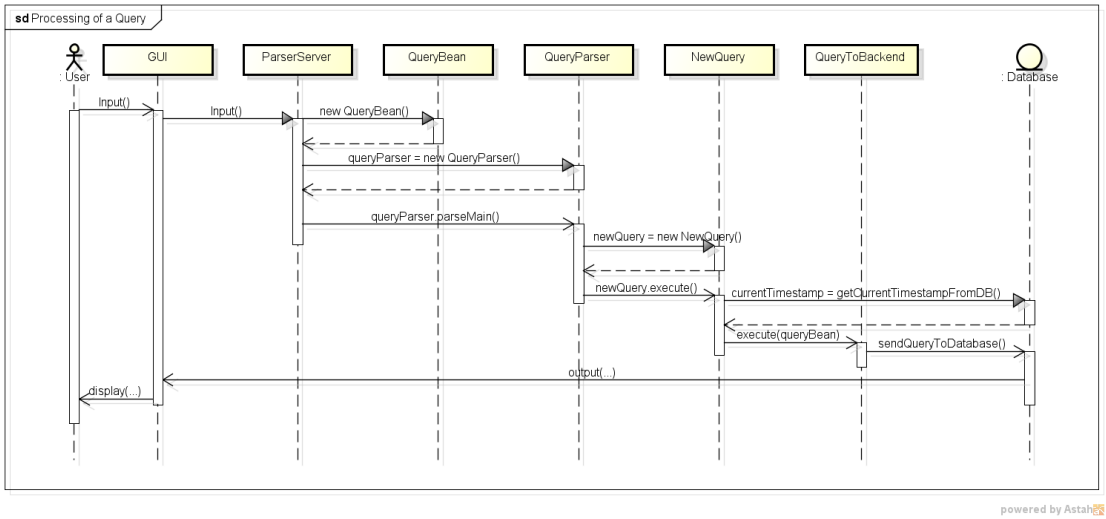

Figure 5.9: Sequence Diagram - Processing of a Query

User

The process starts with the user starting the class “ClientGUI”, entering the required information as well as the query he/she wants to send to the DBMS.

Class ParserServer

When a client connects to the “ParserServer” and a query is sent, an instance of the class “QueryBean” is instantiated as this class stores all the relevant information about the query and the database that will be addressed in the call of this query. Afterwards the class “QueryParser” in the package “org.basex.query” is instantiated and the parsing process is started with the help of the “parseMain()” method.

Class QueryParser

The heart of this class is the “single()” method as it is responsible for the parsing of the query. Due to the fact that this method is called multiple times, we track the state of the parsing process with the variable “singleCalls”. If the number of active “instances” of this method is zero at the end of a “single()” execution, the parsing process is finished.

If an expression of type “insert”, “delete”, “replace”, “replace value” or “rename” is identified while parsing the query, the attribute “newQuery” in the class “QueryBean” is set with a new instance of the respective class (e.g. “NewInsert”, “NewDelete”, etc.). Additionally the attributes “isUpdatingExpression” in the class “QueryBean” is set to “true” and the variable “numberOfUpdatingExpressions” in the same class is incremented by one. If the parser finds another CRUD operation in this query (the number of CRUD operations is already 1), the at-

tribute “errorOccurred” is set to “true” and additionally an “errorMessage” is set (both attributes are inside the class “QueryBean”).

If an expression of type “generateSubset”, “executeAndArchive” or “re-execute” (compare Section 5.4) is found, only the respective boolean “isGenerateSubsetExpression”, “isExecute-AndArchiveExpression” or “isReexecuteExpression” in the class “QueryBean” is set to true.

At the end of the parsing process it is checked whether errors occurred or not. If not, the type of the expression is identified and if it is of type “generateSubset”, “executeAndArchive” or “re-execute” the attribute “newQuery” in the class “QueryBean” is set with a new instance of the respective class. After that the “execute(QueryBean queryBean)” method of the “newQuery” attribute of the class “QueryBean” is executed no matter of which type the expression is.

Class NewQuery

In this class the “execute(QueryBean queryBean)” method creates the queries that have to be sent to the DBMS in order to invoke the needed operations. During this process the method “getCurrentTimestampFromDB()” is executed to retrieve the current system time as timestamp from the database. At this point in time the database is locked with the help of a locking attribute (name: “xmlDB_locking”) in the root node of the query store of the dataset. After the creation of new queries in the class “NewQuery” is done, the queries are sent to the DBMS. This is done by instantiating the class “QueryToBackend” and calling the method “execute(QueryBean queryBean)”.

Class QueryToBackend

This Class sends the queries to the DBMS.

User

At the end of the whole process the user is informed about the result of the query sent to the DBMS.

5.6.8 Graphical User Interface - GUI

The graphical user interface offered to access the server component of the parser is depicted in Figure 5.10. The implementation of this GUI is located in the project “ClientGUI” and can be found unter the path “src/gui/Citation_GUI.java”.

When looking at the GUI it can be seen that the left half serves the purpose of communicating with the parser and the right half visualizes subsets (max. size 50 MB) of the datasets as trees. The white box on the left side is used for printing output like confirmations or error messages to the user. Furthermore metadata about re-executed queries is displayed in this text

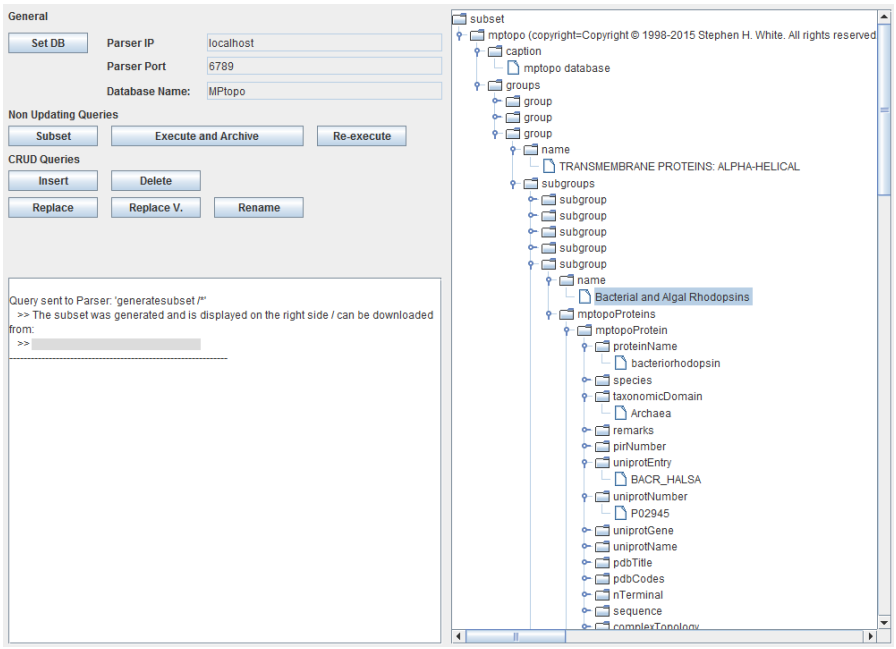

Figure 5.10: GUI of the Client Side of the Parser

area after invoking a re-execute" operation.

After starting the parser the user has to set the IP of the parser, its port and the name of the database that needs to be accessed by using the "Set DB" button in order to unlock the other buttons and make interactions with the parser possible. The three buttons under the label "Non Updating Queries" offer the possibility to invoke the query types "generateSubset" ("Subset" button), "executeAndArchive" and "re-execute".

The five buttons grouped by the "CRUD Operations" label help users to build queries invoking CRUD operations. When clicking one of these buttons a small query building assistant is started. This means e.g. for the "insert" operation the user can decide whether an element, an attribute or a text node is to be inserted. In the case of the insertion of an attribute the user is asked to insert the name of the new attribute, its value and the path to the element node the attribute has to be inserted into. The respective query is built by the GUI in the background and is sent to the parser automatically.

5.7 Summary

In this chapter we presented a way to to overcome the problem of data citation in the context of XML databases by using one of the two presented approaches. After having pointed out which prerequisites were identified and assumptions were made, we described the parser that

we implemented. Besides the architecture of the parser, we also presented components that are needed in order to work with it.

CHAPTER 6

Evaluation of the Versioning and Subsetting Solution

This chapter serves the purpose of evaluating the approaches presented by us in Chapter 5. Starting with Section 6.1, the system specifications will be listed. Section 6.2 will explain how both approaches are evaluated and Section 6.3 will deal with performance tests of both approaches and compares them.

As we have already stated in the introduction, all the documents that are described in this chapter can be downloaded from the homepage `http://www.ifs.tuwien.ac.at/dp/datacitation_xml`.

6.1 System Specifications

For the experiments that are conducted in this thesis, the system specified in Table 6.1 was used. In order to ensure a stable system performance we disabled all the non essential background programs.

6.2 Evaluation of the Functionality of Approach I and II

The functionalities of both approaches that we presented in Chapter 5 were evaluated with the help of manual system tests. A list of all executed queries invoking the CRUD operations “insert”, “delete”, “replace”, “replace value” and “rename” can be downloaded from the homepage specified at the beginning of this chapter.

Table 6.1: System Specifications

Hardware	
CPU	Intel Core i3-4030U @ 1.90 GHz
RAM	4096 MB
GPU	Intel HD Graphics 4400
Software	
Windows	Windows 7 Professional x64
Java JDK	1.8.0_45
Database: "BaseX"	Version 8.2.3
Database: "eXist-db"	Version 2.2

6.3 Performance

In this section we will take a closer look at the performance of both approaches when they are run on different database management systems. Two scenarios, one of them only inserting new pieces of information and the other one focusing on updating information will be run for both of the two approaches. For measuring the performance we use the two artificial datasets described in Section 3.5.1 that were created by us. Two versions, namely one with deep hierarchy and one with flat hierarchy, were created. Due to the fact that the numbers of leaf nodes (text nodes) are the same, we assume both datasets to contain an equals amount of information although they differ in size.

CRUD operations are only performed on the lower half (rounded up) of the XML datasets because we identified that updates on high hierarchical levels are unlikely in real world datasets. When talking about the dataset with flat hierarchy this means that updates are performed on the element nodes "level2", "level3", "level4" and their respective attributes and text nodes. For the dataset with deep hierarchy, the editable elements are all the nodes from "level6" to "level12" including their attributes and texts. An overview over these nodes and the probabilities of them being selected for updates is given in Figure 6.1.

Four sets of queries were produced - one for the dataset with deep and one for the dataset with flat hierarchy multiplied by two as two scenarios "updating" and "inserting" are to be differentiated. Tracking the generated queries ensures that the same set of queries can be executed on all the combinations of "BaseX"/"eXist-db" and Approach I - "Branch Copy"/Approach II - "Parent-Child" for the respective type of dataset. This means that the execution times of the CRUD operations as well as the resulting sizes of datasets can be compared easily.

Generally it can be said that the time that is needed to process the queries sent to the DBMS are measured. This is done by tracking the current system time at which the database is locked (insertion of the locking attribute into the root node of the query store) and unlocked again (deletion of the locking attribute). These two timestamps are saved in the class "QueryBean.java" in the two attributes "timestampStart" and "timestampEnd". After the query is finished, its runtime is measured based on these two attributes. As a last step, information about the query is stored in CSV files. The two types of CSV files that are maintained will be described at the end of this

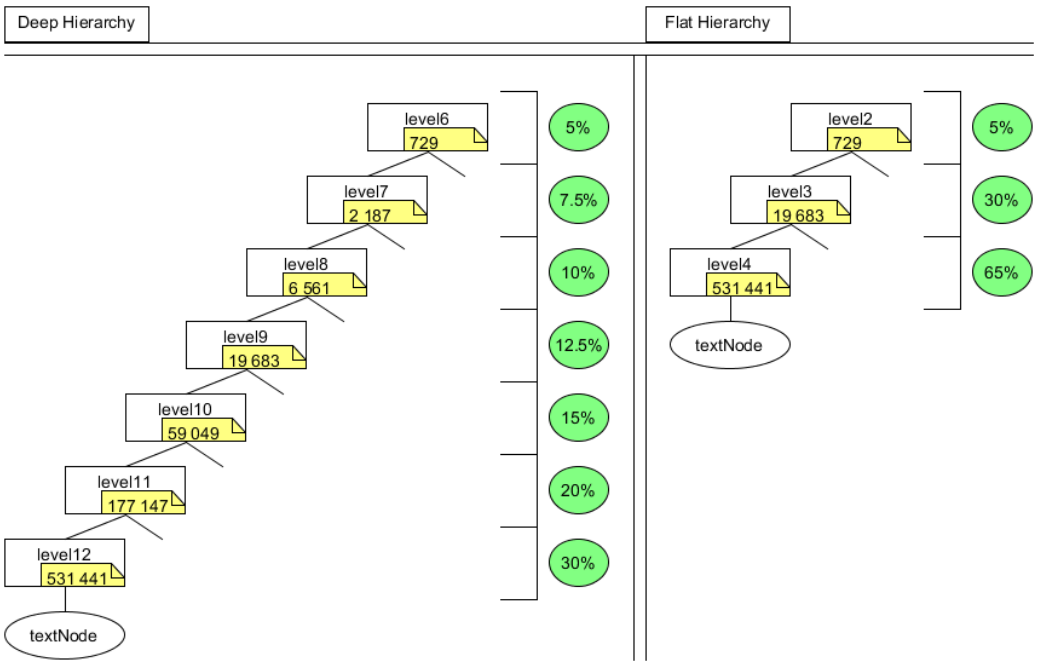

Figure 6.1: Editable Nodes

section.

We created a random CRUD operation generator that directly sends the operations to the parser after generating them. After determining the level an update has to be performed on a random node is chosen. Finally a CRUD operation is chosen, a query is formulated and the query is sent to the parser. When testing a dataset with deep or flat hierarchy for the first time, queries are generated but if queries already exist, no new ones are created but the already existing ones are executed in order to make comparing of the results possible.

The most common basic database operations are called CRUD operations [32]. These operations include Create, Retrieve, Update and Delete statements. Thus these four operations are the most important when it comes to persistent storage. Due to this fact, the generated updates created by the our CRUD operation generator are based on these four basic operations.

Table 6.2 depicts which CRUD operations are possible and how likely they are to appear in the respective scenario. The operation of replacing a text with a text is disabled because of the fact that replacing the value of a text with another text node yields the exact same results. Furthermore the operations of replacing the value of an element with a text is disabled as well because of the fact that replacing a text node with another text yields the same results when

Table 6.2: Possible CRUD Operations in the Respective Scenarios

Scenario	**Updating**				**Inserting**			
Hierarchy	**Deep**		**Flat**		**Deep**		**Flat**	
	No Leaf (Level 6-11) w. 70%	**Leaf (Level 12) w. 30%**	**No Leaf (Level 2-3) w. 35%**	**Leaf (Level 4) w. 65%**	**No Leaf (Level 6-11) w. 70%**	**Leaf (Level 12) w. 30%**	**No Leaf (Level 2-3) w. 35%**	**Leaf (Level 4) w. 65%**
Insert	**40%**	**40%**	**40%**	**40%**	**100%**	**100%**	**100%**	**100%**
Elem into Elem	20%	—	20%	—	80%	—	80%	—
Attr into Elem	20%	20%	20%	20%	20%	20%	20%	20%
Text into Elem	—	20%	—	20%	—	80%	—	80%
Delete	**15%**	**15%**	**15%**	**15%**	**—**	**—**	**—**	**—**
Elem	5%	5%	5%	5%	—	—	—	—
Attr	10%	5%	10%	5%	—	—	—	—
Text	—	5%	—	5%	—	—	—	—
Replace	**5%**	**5%**	**5%**	**5%**	**—**	**—**	**—**	**—**
Elem with Elem	5%	5%	5%	5%	—	—	—	—
Replace Value	**30%**	**30%**	**30%**	**30%**	**—**	**—**	**—**	**—**
Attr with Text	30%	20%	30%	20%	—	—	—	—
Text with Text	—	10%	—	10%	—	—	—	—
Rename	**10%**	**10%**	**10%**	**10%**	**—**	**—**	**—**	**—**
Elem as Text	5%	5%	5%	5%	—	—	—	—
Attr as Text	5%	5%	5%	5%	—	—	—	—

performing the “replace value of element with text” operation only on leaf element nodes.

When looking at Table 6.2 it can be seen that we distinguish between updates on level 2-3 for flat hierarchy datasets / level 6 - level 11 for deep hierarchy datasets and level 4 / 12. This was done because element nodes on the deepest hierarchical level have only a single text and no other element nodes as child. As a result other CRUD operations like the insertion or updating of text nodes can be performed on this level. When it comes to element nodes on higher hierarchical levels these operations are not possible due to the fact that simply no text nodes exist. On these levels other operations like the insertion of element nodes are made possible.

The course of action of the performance tests is that at the start a couple of queries are archived with the help of the “executeAndArchive” operation. Afterwards 10 000 CRUD operations are executed updating the state of the database. With ongoing updates a change in execution times as well as a change in size of both the up to date as well as the history database can be observed. At the end of all the updates re-execution times can be measured and compared as well.

CSV Files Containing only Queries

Four files of this type are generated in the course of the performance measurements, namely “Performance_deep_Queries_Updating” and “Performance_flat_Queries_Updating” (“Performance_deep_Queri

and “Performance_flat_Queries_Inserting” for the “inserting” scenario) holding only the executed queries. In detail this means that the “operation” (e.g. “insert”, “delete”, etc.), the “combination” of node types (e.g. “ElementIntoElement” for the “insert” operation), the “level” (hierarchical level the update was performed on) and the complete “query” are stored.

CSV Files Containing Time Records

This file holds the same information as the already described file but additionally adds “start” and “end” representing the starting and ending time in the form of milliseconds. As a last piece of information “totalTime” representing the time the query took for execution is included.

For every combination of “BaseX”/“eXist-db”, Approach I/Approach II, deep/flat and “updating”/“inserting” scenario one file is generated. The naming of the files is stated in Listing 6.1.

Listing 6.1: Naming of CSV Files That Contain Time Records

```
Performance_[flat/deep]Approach[1/2][BaseX/EXist][Inserting/
   Updating].csv
```

6.3.1 Sizes of Datasets

The evolution of the sizes of the databases is depicted in Figure 6.2 for the deep hierarchy dataset and in Figure 6.3 for the flat hierarchy dataset. It can be seen that the history database when archiving with Approach I - “Branch Copy” becomes very big in size compared to the up to date database. The big difference in size of the history database between Approach I - “Branch Copy” and Approach II - “Parent-Child” is a result of every element node containing two timestamp attributes showing their validity when it comes to Approach I - “Branch Copy”. Furthermore whole branches are copied in the case of a versioning with Approach I - “Branch Copy” whereas Approach II - “Parent-Child” only uses a few timestamps including versioning blocks to show the validity of nodes.

When looking at the “updating” scenario one can see that the up to date database shrinks over time. When an element is replaced with another element node the old element including its whole subtree gets deleted and the newly inserted element node has only got one child per hierarchical level on its “descendant” axis. This means that when an element node on a high hierarchical level is to be replaced, the total number of element nodes in the database is lowered. As a result the size of the database shrinks.

Comparing the sizes of the up to date databases to the ones of the respective history database when archiving with Approach II - “Parent-Child” it can be seen that they hardly differ. In order to highlight the long time behaviour of this approach Figures 6.4 and 6.5 depict the development of the sizes of the databases for the “inserting” scenario. For this analysis 100 000 CRUD operations were run. The “updating” scenario was not evaluated because of just stated problem

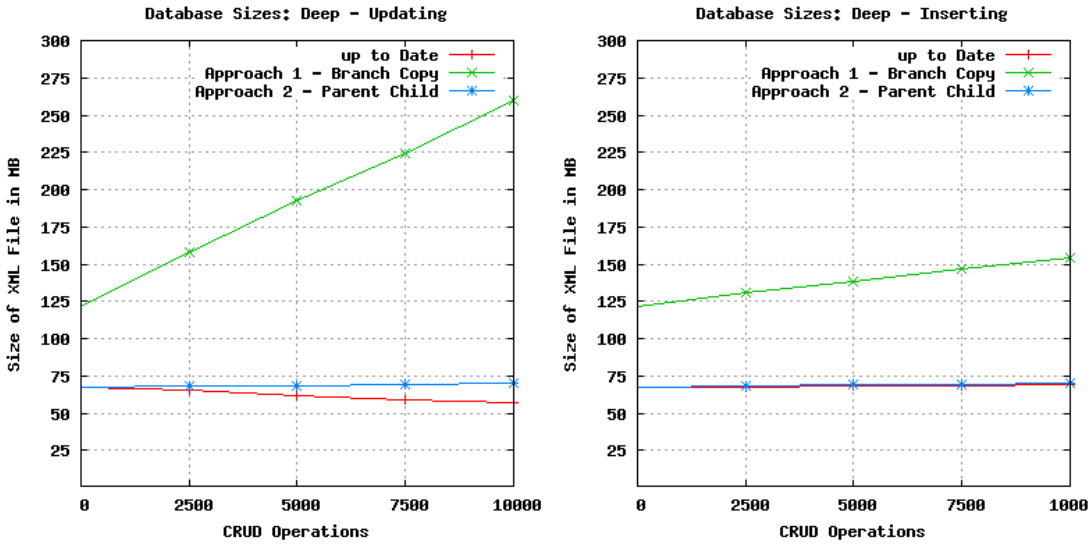

Figure 6.2: Size of the Deep Hierarchy Dataset

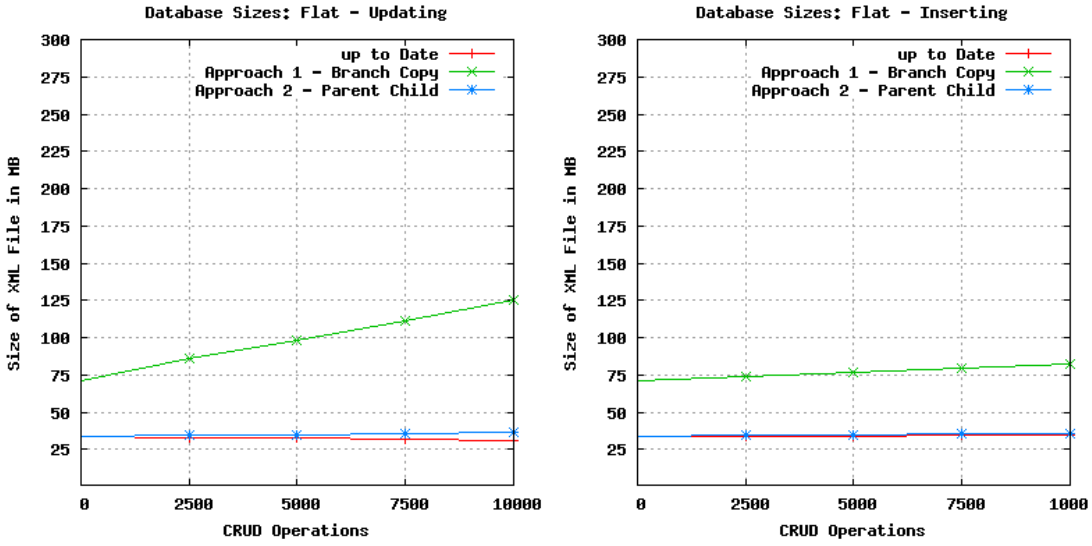

Figure 6.3: Size of the Flat Hierarchy Dataset

of the shrinking up to date database. This would have resulted in a very small up to date database and was thus not included.

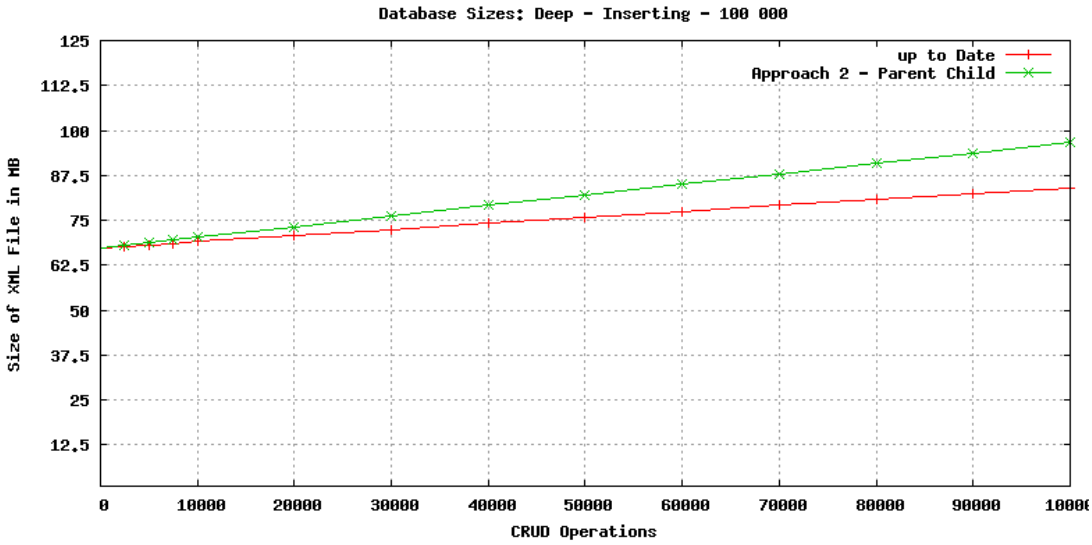

Figure 6.4: Size of the Deep Hierarchy Dataset - Approach II - Inserting Scenario - DBMS: BaseX - Long Run

6.3.2 Query Performance

The performances of the CRUD operations for the DBMS "BaseX" are visualised in Figures 6.6 and 6.7. When taking a closer look at the graphs it can easily be seen that Approach II - "Parent-Child" performes better than Approach I - "Branch Copy". Furthermore it can be seen that especially the execution times for the deep hierarchy dataset in the "updating" scenario when archiving with Approach I - "Branch Copy" differ strongly because of the fact that whole branches on different hierarchical levels have to be copied.

An interesting observation we made is that BaseX is not fully consistent concerning the execution times of the insertion of element nodes. The performance spikes, which can be seen for example in the lower graph of Figure 6.7, mainly relate to the insertion of element nodes into elements. Anyways when taking a look at the average query performance it becomes clear that Approach II - "Parent-Child" performs better than Approach I - "Branch Copy".

Concerning the DBMS "eXist-db" we discovered a very slow performance, which is depicted in Figure 6.8. For the "inserting" scenario and the dataset with flat hierarchy updates took between 9 and 15 seconds instead of around 0.5 seconds when using the DBMS "BaseX". Although we set range indices [1] and tried to tune the database management systems performance [2] we could not achieve query times that were able to compete with the ones of "BaseX". Hence

[1] Compare `http://exist-db.org/exist/apps/doc/newrangeindex.xml` - Last access: 27-08-2015

[2] Compare `http://exist-db.org/exist/apps/doc/tuning.xml` - Last access: 27-08-2015

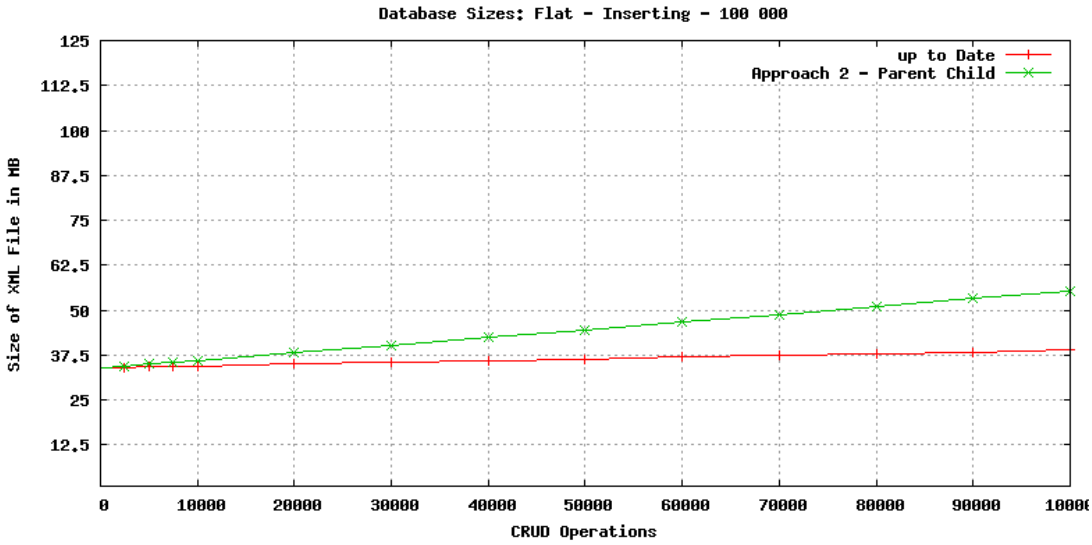

Figure 6.5: Size of the Flat Hierarchy Dataset - Approach II - Inserting Scenario - DBMS: BaseX - Long Run

we put “eXist-db” aside and focused solely on “BaseX”.

Because of the same reasons that we have already identified in Section 6.3.1 we leave Approach I - “Branch Copy” aside and evaluated the query performance in the long run for the flat as well as the deep hierarchy dataset in the “inserting” scenario. In the course of these performance measurements we found out that with the time the query performance gets worse and worse until “BaseX” throws an error caused by problems with bad indexing. As a first solution we used the configuration “UPDINDEX”, which is offered by “BaseX” and optimizes the databases after each update. The diagram depicted in Figure 6.9 shows that the runtimes increased drastically with this configuration so we quit the measurement after 20 000 operations.

In order to keep indexes up to date and to ensure that the performance is hardly influenced by the index structure of “BaseX” we executed an optimization command and thus rebuilt the databases after 2 500 operations each. The figures 6.10 and 6.11 depict the measurements we made with the help of this optimization command. Again it can be seen that BaseX is not consistent when it comes to the time it takes in order to insert element nodes into elements. Additionally it can be seen that the execution of the first query that is invoked after the optimization of the databases takes significantly longer. However, it can also be seen that the average runtime of queries increases by around 200 ms after 100 000 CRUD operations.

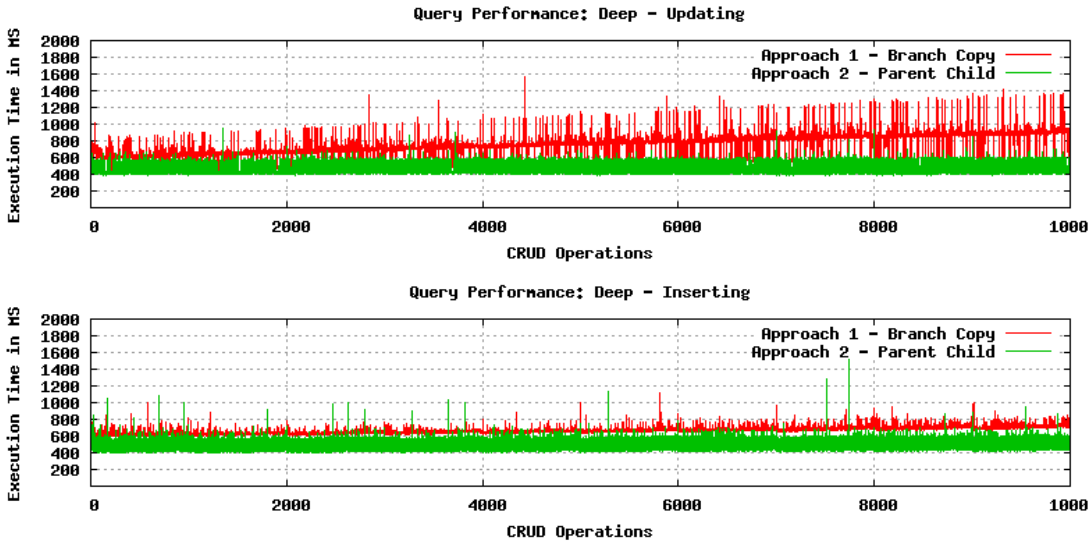

Figure 6.6: Query Performance of the Deep Hierarchy Dataset - DBMS: BaseX

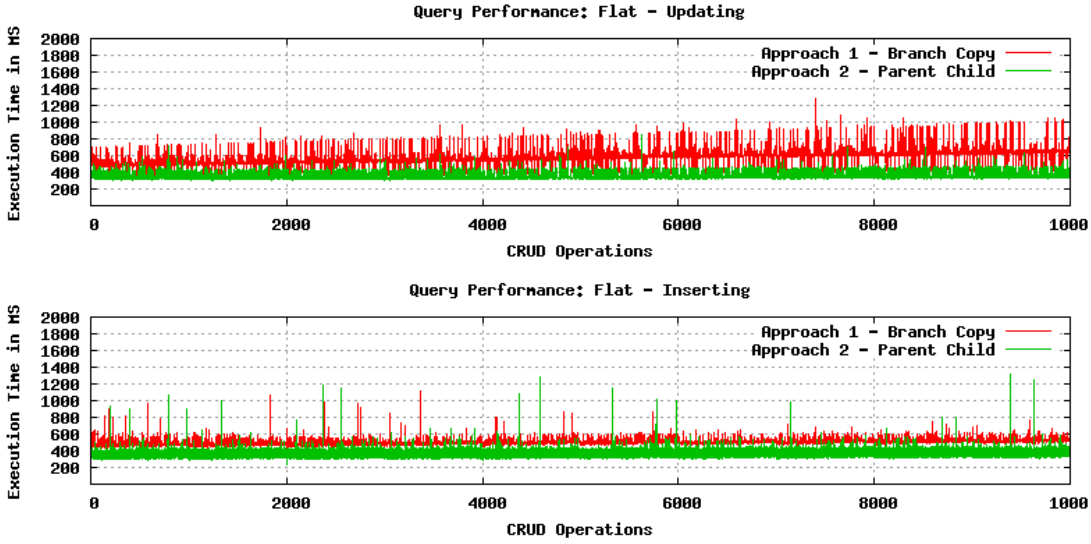

Figure 6.7: Query Performance of the Flat Hierarchy Dataset - DBMS: BaseX

6.3.3 "ExecuteAndArchive" and "Re-execute" Query Performance

An important factor when deciding whether the presented approaches are viable or not is the time the parser needs to re-execute a stored query and thus to return a certain subset. For both the deep hierarchy dataset as well as the flat hierarchy dataset five select queries (listed in Appendix 1) were archived at the beginning and re-executed after performing 10 000 CRUD operations.

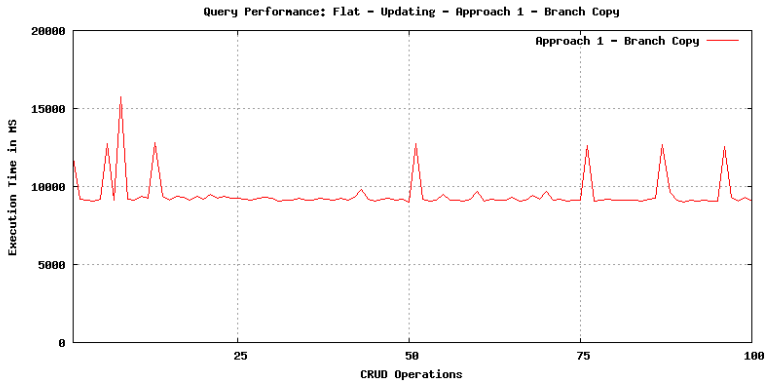

Figure 6.8: Query Performance of the Flat Hierarchy Dataset - DBMS: eXist-db

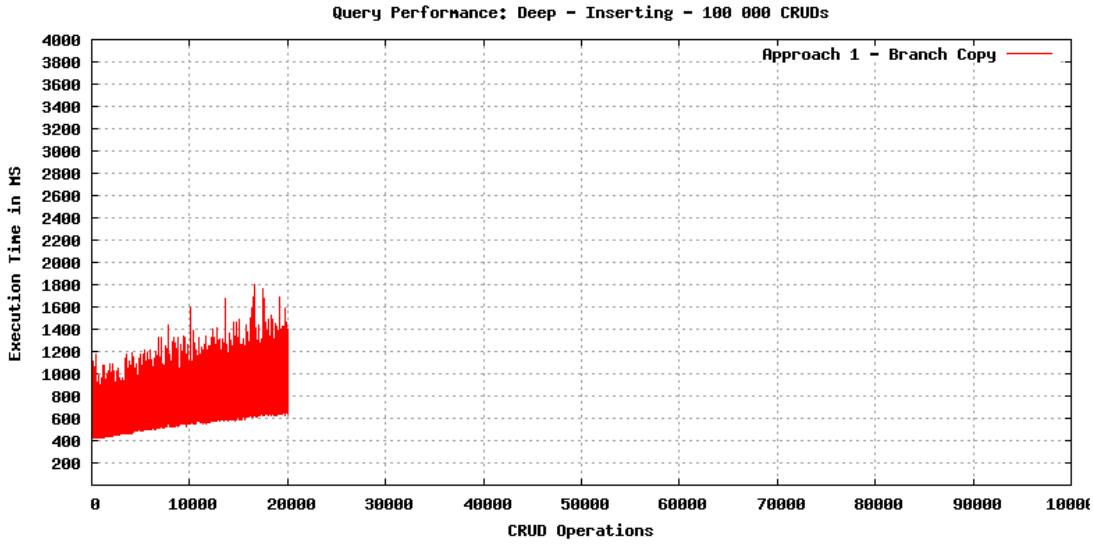

Figure 6.9: Query Performance of the Flat Hierarchy Dataset - Approach II - Inserting Scenario - Long Run - DBMS: BaseX - UPDINDEX

The select queries were written in a way so that they return an equal amount of text nodes for the deep and the flat hierarchy dataset. This means that e.g. the first query for the flat hierarchy dataset returns an equals amount of text nodes as the first query for the deep hierarchy dataset. Table 6.3 depicts the amount of text nodes identified per query.

Table 6.4 highlights the seconds it takes in order to re-execute the five defined queries after 10 000 CRUD operations. Again it can be seen that Approach II - “Parent-Child” outperformes Approach I - “Branch Copy” by far. The re-execution times after 100 000 CRUD operations and are depicted in Table 6.5. It can be seen that the time it takes to re-execute an archived query

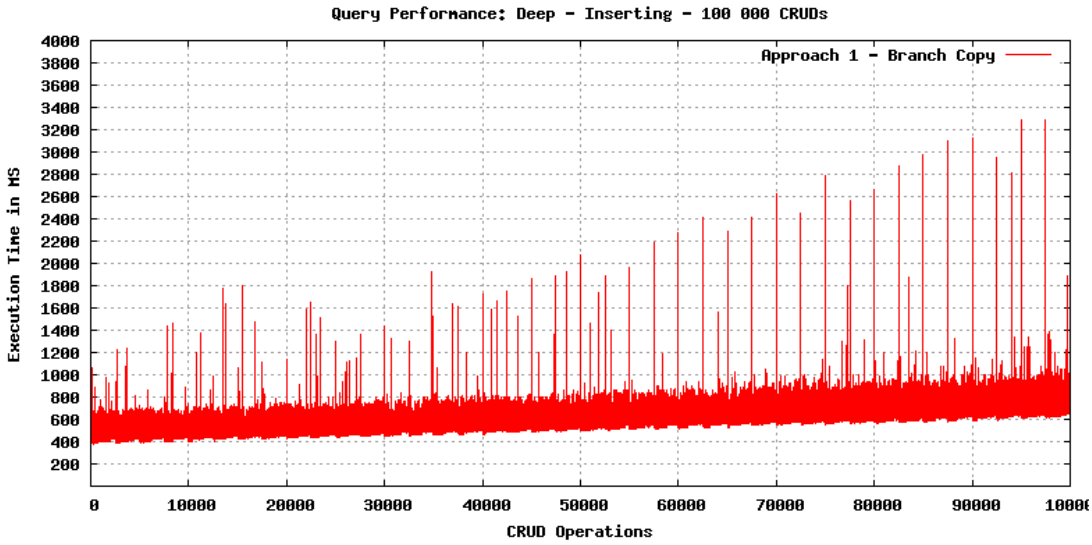

Figure 6.10: Query Performance of the Deep Hierarchy Dataset - Approach II - Inserting Scenario - DBMS: BaseX - Long Run

Table 6.3: Number of Selected Text Nodes for "select" Queries

Select Number	Number of Text Nodes	Level - Deep	Level - Flat
Select 1	531 441	Level0	Level0
Select 2	19 683	Level3	Level1
Select 3	729	Level6	Level2
Select 4	27	Level9	Level3
Select 5	1	Level12	Level4

after 100 000 updates roughly doubles compared to the re-execution times after 10 000 updates.

6.3.4 Short comparison to XArch

As the versioning techniques of XArch and our approaches are very different they are hard to compare. XArch builds up an archive where database dump have to be merged together. Subsets are derived with the help of an own query language. Our approaches on the other hand archive datasets directly inside the database. Subsets can be queried with the help of normal XPath and XQuery.

As we tried out XArch we experienced a high performance when identifying and deriving subsets. However, users that want to use this type of versioning mechanism have to formulate their queries in a different language. Furthermore attribute nodes are transformed into an internal node structure that is similar to element nodes in XML. Thus the structure of the document gets lost. Transforming attributes into an element like structure increases the performance and

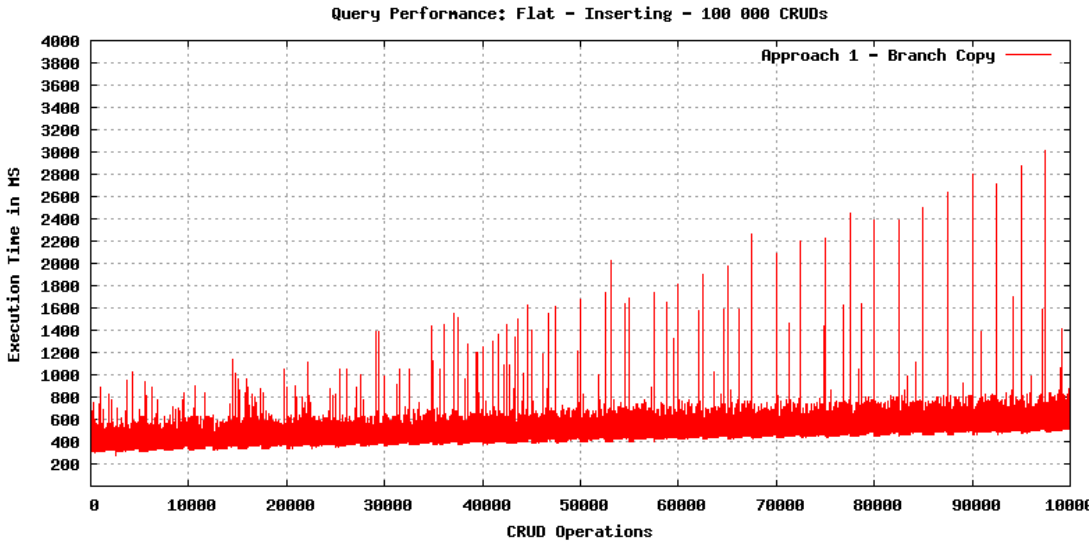

Figure 6.11: Query Performance of the Flat Hierarchy Dataset - Approach II - Inserting Scenario - DBMS: BaseX - Long Run

Table 6.4: Re-execution Performance (in Seconds) after 10 000 CRUD Operations - DBMS: BaseX

Deep Select Query	Approach I - "Branch Copy" Updating	Inserting	Approach II - "Parent-Child" Updating	Inserting
Select 1	22.68	19.02	6.05	5.60
Select 2	20.90	17.12	4.02	3.54
Select 3	21.00	17.19	4.14	3.78
Select 4	21.66	16.52	4.09	3.57
Select 5	22.58	17.68	4.50	3.95
Flat Select Query	**Approach I - "Branch Copy" Updating**	**Inserting**	**Approach II - "Parent-Child" Updating**	**Inserting**
Select 1	13.42	12.83	4.39	4.28
Select 2	11.39	10.89	3.07	2.71
Select 3	11.28	10.36	2.96	2.61
Select 4	10.89	10.26	2.84	2.56
Select 5	11.27	11.18	3.08	2.82

thus we expect our approaches to never perform better than XArch.

The combination of both ways of archiving - so to say storing different versions of the database in XArch, creating a database in a native XML database management system and executing XPath/XQuery queries in that environment would be too much of a performance loss. Taking a look at the "re-execute" performance of Approach II - "Parent-Child" shows us that the creation of a new database takes more time than actually re-executing" archived queries.

Table 6.5: Re-execution Performance (in Seconds) after 100 000 CRUD Operations - Approach II - Inserting Scenario - DBMS: BaseX - Long Run

Deep Select Query	**After 10 000 Operation**	**After 100 000**
Select 1	5.60	9.75
Select 2	3.54	7.30
Select 3	3.78	7.58
Select 4	3.57	7.18
Select 5	3.95	7.62
Flat Select Query	**After 10 000 Operation**	**After 100 000**
Select 1	4.28	7.35
Select 2	2.71	5.71
Select 3	2.61	5.75
Select 4	2.56	5.42
Select 5	2.82	5.67

CHAPTER 7

Summary and Future Work

In the course of this thesis we dealt with the topic of making subsets of XML datasets in native XML databases citable. After having discussed the theoretical groundwork for this thesis we introduced the concept of a parser that acts as a middleware between the user and the DBMS and enables the versioning of subsets. This can be done with the help of two different approaches - both of which have been developed and implemented by us. Concerning the structure of the database it can be said that the dataset exists in two versions - one being up to date all the time and the other one containing all the version histories. Depending on whether the up to date dataset has to be queried or a query has to be re-executed the respective database is chosen by the parser.

In the chapter of dealing with evaluation we found that Approach I - "Branch Copy" does not scale in terms of query performance and required storage space as whole branches are copied and thus the history database grows to an enormous size compared to the up to date database. As a result this approach was turned down and not tested any further. In contrast we identified that Approach II - "Parent Child" offers a decent performance of around 0.5 to 0.7 seconds processing time for a CRUD operation after having already invoked 100 000 CRUD statements. In terms of memory consumption the "Parent Child" turned out to be very efficient as archiving with the help of this approach only roughly doubles the size of the database that has to be archived (up to date + history database). In fact the history database will grow bigger than the up to date database as information is never deleted but the difference between the sizes of these two databases is quite small (around 5 MB difference after the execution of 100 000 CRUD updates depending on the operations being executed).

Due to necessary limitations and restrictions of this thesis we could not address all the interesting aspects within our field of research. These optimizations will be interesting to look at in future work.

The first crucial topic to be listed is the performance of the "re-execution" function of the parser. Currently the whole history database is copied and its old state is constructed. If the

subset that has to be identified covers most parts of the dataset, this is an efficient solution but if only a small subset is to be derived unnecessary transformations are invoked. So this point is definitely an important one showing potential for improvements. Furthermore we think that queries that are written and invoked by the parser can be improved in terms of performance. When it comes to query performance a lot of factors like the structure of a query, the DBMS in the backend and even the configuration of the DBMS play a role.

Another feature that is currently not covered by the parser presented in this thesis is the handling of concurrent operations. In this state the communication between the client and the server is based on a simple socket connection which makes concurrency topics hard. A point being connected to the topic of concurrency is a proper rule when generated subsets are to be deleted from the server. Furthermore no query queue or something like that is implemented at the moment. Concerning the technical implementation of the parser there is also potential when it comes to user management.

The last point being identified as topic for future work is the usage of a namespace for attributes and elements that are only used for the versioning of XML nodes. Currently the user is restricted and must not use the keyword “xmlDB_” because it is reserved for the parser.

All of the above mentioned improvements will need to be dealt with in future work for which the present thesis and its results serve as a basis. This thesis could suggest two solutions to the problem of making XML databases citable of which one is definitely worth looking at in more detail.

APPENDIX 1

ExecuteAndArchive Queries

The Listings 1 .1 - 1 .5 state the “executeAndArchive” queries for the dataset with deep hierarchy and Listings 1 .6 - 1 .10 show the queries for the dataset with flat hierarchy.

Deep Hierarchy

Listing 1 .1: Hierarchy: Deep - Target: One “Level0” Element - Identified Text Nodes: 531 441

```
executeandarchive /level0
```

Listing 1 .2: Hierarchy: Deep - Target: One “Level3” Element - Identified Text Nodes: 19 683

```
executeandarchive /level0/level1[@id="0"]/level2[@id="0"]/
   level3[@id="0"]
```

Listing 1 .3: Hierarchy: Deep - Target: One “Level6” Element - Identified Text Nodes: 729

```
executeandarchive /level0/level1[@id="2"]/level2[@id="2"]/
   level3[@id="2"]/level4[@id="2"]/level5[@id="2"]/
   level6[@id="2"]
```

Listing 1 .4: Hierarchy: Deep - Target: One “Level9” Element - Identified Text Nodes: 27

```
executeandarchive /level0/level1[@id="1"]/level2[@id="1"]/
   level3[@id="1"]/level4[@id="1"]/level5[@id="1"]/
   level6[@id="1"]/level7[@id="1"]/level8[@id="1"]/
   level9[@id="1"]
```

Listing 1 .5: Hierarchy: Deep - Target: One “Level12” Element - Identified Text Nodes: 1

```
executeandarchive /level0/level1[@id="1"]/level2[@id="1"]/
```

```
   level3[@id="1"]/level4[@id="1"]/level5[@id="1"]/
   level6[@id="1"]/level7[@id="1"]/level8[@id="1"]/
   level9[@id="1"]/level10[@id="1"]/level11[@id="1"]/
   level12[@id="1"]
```

Flat Hierarchy

Listing 1 .6: Hierarchy: Flat - Target: One “Level0” Element - Identified Text Nodes: 531 441

```
executeandarchive /level0
```

Listing 1 .7: Hierarchy: Flat - Target: One “Level1” Element - Identified Text Nodes: 19 683

```
executeandarchive /level0/level1[@id="0"]
```

Listing 1 .8: Hierarchy: Flat - Target: One “Level2” Element - Identified Text Nodes: 729

```
executeandarchive /level0/level1[@id="0"]/level2[@id="0"]
```

Listing 1 .9: Hierarchy: Flat - Target: One “Level3” Element - Identified Text Nodes: 27

```
executeandarchive /level0/level1[@id="0"]/level2[@id="0"]/
   level3[@id="0"]
```

Listing 1 .10: Hierarchy: Flat - Target: One “Level4” Element - Identified Text Nodes: 1

```
executeandarchive /level0/level1[@id="0"]/level2[@id="0"]/
   level3[@id="0"]/level4[@id="0"]
```

Bibliography

[1] Serge Abiteboul, Peter Buneman, and Peter Suciu. *Data on the Web - From Relations to Semistructured Data and XML*. Morgan Kaufmann, San Francisco, Calif, 2000.

[2] Timothy T. Allen. Citing References in Scientific Research Papers, 2000. `http://tim.thorpeallen.net/Courses/Reference/Citations.pdf`. Last access: 20-09-2013.

[3] Micah Altman and Gary King. A Proposed Standard for the Scholarly Citation of Quantitative Data. *D-Lib Magazine*, 13(3/4), March / April 2007.

[4] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. A Data Model for Temporal XML Documents. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 334–344, London, UK, 2000. Springer-Verlag.

[5] Mitch Amiano, Kay Ethier, Conrad D'Cruz, and Michael D. Thomas. *XML: Problem - Design - Solution*. Programmer to programmer. Wiley, 2006.

[6] Arbortext Inc, Sun Microsystems Inc. Axis Specifiers, 2000. `http://nwalsh.com/docs/tutorials/xsl/xsl/foil22.html`. Last access: 11-09-2014.

[7] Australian National Data Service. Data Citation, 2011. `http://ands.org.au/guides/data-citation-awareness.pdf`. Last access: 15-11-2014.

[8] Emanuele Bellini, Chiara Cirinnà, Maurizio Lunghi, Ernesto Damiani, and Cristiano Fugazza. Persistent Identifiers Distributed System for Cultural Heritage Digital Objects, 2008.

[9] Emanuele Bellini, Cinzia Luddi, Chiara Cirinna, Maurizo Lunghi, Achille Felicetti, Barbara Bazzanella, and Paolo Bouquet. Interoperability Knowledge Base for Persistent Identifiers Interoperability Framework. In *Eighth International Conference on Signal Image Technology and Internet Based Systems (SITIS), 2012*, pages 868–875, 2012.

[10] Bert Bos. The XML Data Model, 2005. `http://www.w3.org/XML/Datamodel.html`. Last access: 25-08-2015.

[11] Ronald Bourret. Going Native: Use Cases for Native XML Databases, 2007. http://www.rpbourret.com/xml/UseCases.htm. Last access: 26-02-2015.

[12] Ronald Bourret. Xml Database Products: XML-Enabled Databases, 2010. http://www.rpbourret.com/xml/ProdsXMLEnabled.htm. Last access: 25-02-2015.

[13] Neil Bradley. *The XML Companion*. Addison-Wesley Professional, Boston, 3 sub edition, 2002.

[14] Jan Brase. DataCite - A Global Registration Agency for Research Data. In *Fourth International Conference on Cooperation and Promotion of Information Resources in Science and Technology, 2009*, pages 257–261, 2009.

[15] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. Archiving Scientific Data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 1–12, New York, NY, USA, 2002.

[16] Peter Buneman, Ioannis Koltsidas, Heiko Müller, and Stratis Viglas. Xarch the <XML> Archiver, 2015. http://xarch.sourceforge.net/index.html. Last access: 01-08-2015.

[17] Akmal B. Chaudhri, Awais Rashid, and Roberto Zicari. *XML Data Management - Native XML and XML-enabled Database Systems*. Addison-Wesley Professional, Boston, 2003.

[18] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change Detection in Hierarchically Structured Information. *SIGMOD Rec.*, 25(2):493–504, June 1996.

[19] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting Changes in XML Documents. In *Proceedings of the 18th International Conference on Data Engineering, 2002*, pages 41–52, 2002.

[20] Frank P. Coyle. *XML, Web Services, and the Data Revolution*. Addison-Wesley information technology series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[21] Robert Darby, Simon Lambert, Brian Matthews, Michael Wilson, Kathrin Gitmans, Suenje Dallmeier-Tiessen, Salvatore Mele, and Jari Suhonen. Enabling Scientific Data Sharing and Re-use. In *IEEE 8th International Conference on E-Science (e-Science), 2012*, pages 1–8, 2012.

[22] Kevin Dick. *XML: A Manager's Guide*. Addison-Wesley information technology series. Addison-Wesley, 2003.

[23] Ixchel M. Faniel and Ann Zimmerman. Beyond the Data Deluge: A Research Agenda for Large-Scale Data Sharing and Reuse. *International Journal of Digital Curation*, 6(1):58–69, 2011.

[24] Joe Fawcett, Danny Ayers, and Liam R. E. Quin. *Beginning XML, 5th Edition*. John Wiley & Sons, New York, 2012.

[25] Anne M. Fitzgerald, Kylie M. Pappalardo, Brian F. Fitzgerald, Anthony C. Austin, John W. Abbot, Brendan L. Cosman, Damien S. O'Brien, and Bill Singleton. Building the Infrastructure for Data Access and Reuse in Collaborative Research : An Analysis of the Legal Context. The OAK Law Project, June 2007.

[26] Sergey Fomel and Jon F. Claerbout. Guest Editors' Introduction: Reproducible Research. *Computing in Science Engineering*, 11(1):5–7, Jan 2009.

[27] German National Library. Persistent Identifiers, 2012. `http://www.dnb.de/EN/Standardisierung/PI/pi`. Last access: 12-09-2013.

[28] P. S. Gill. *Database Management Systems*. I. K. International Pvt Ltd, 2008.

[29] David Gulbransen, Kynn Bartlett, and Earl Bingham. *Using XML*. Que Publishing, Indianapolis, 2002.

[30] Beda Christoph Hammerschmidt. *KeyX - Selective Key-oriented Indexing in Native XML-databases*. IOS Press, München, 2006.

[31] Elliotte Rusty Harold and Scott W. Means. *XML in a Nutshell*. O'Reilly Media, Inc., Sebastopol, CA, 2004.

[32] Martin Heller. Rest and CRUD: the Impedance Mismatch, 2007. `http://www.infoworld.com/d/developer-world/rest-and-crud-impedance-mismatch-927`. Last access: 28-04-2014.

[33] Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The Fourth Paradigm - Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[34] Hans-Werner Hilse and Jochen Kothe. *Implementing Persistent Identifiers: Overview of Concepts, Guidelines and Recommendations*. Consortium of European Research Libraries, European Commission on Preservation and Access, 2006.

[35] International DOI Foundation. DOI System and Persistent URLs (PURLs), 2015. `http://www.doi.org/factsheets/DOI_PURL.html`. Last access: 23-08-2015.

[36] Sajith Jayasinghe, Kalina Hristova, and Stephen H. White. MPtopo: A Database of Membrane Protein Topology. *Protein Science*, 10(2):455–458, 2001.

[37] Howard Katz, Don Chamberlin, and Dennis Draper. *XQuery from the Experts - A Guide to the W3C XML Query Language*. Addison-Wesley Professional, Boston, 2004.

[38] Seema Kedar. *Database Management System*. Technical Publications, 2009.

[39] Peter Klahold, Gunter Schlageter, and Wolfgang Wilkes. A General Model for Version Management in Databases. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 319–327, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[40] Jelena Kovacevic. How to Encourage and Publish Reproducible Research. In *IEEE International Conference on Acoustics, Speech and Signal Processing. ICASSP 2007.*, volume 4, pages IV–1273–IV–1276, 2007.

[41] Krishna Kulkarni and Jan-Eike Michels. Temporal Features in SQL:2011. *SIGMOD Rec.*, 41(3):34–43, October 2012.

[42] Wilburt Labio and Hector Garcia-Molina. Efficient Snapshot Differential Algorithms for Data Warehousing. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 63–74, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[43] Steve Lawrence, Frans Coetzee, Eric Glover, David Pennock, Gary Flake, Finn Nielsen, Bob Krovetz, Andries Kruger, and Lee Giles. Persistence of Web References in Scientific Research. *IEEE Computer*, 34(2):26–31, 2001.

[44] Wangchao Le, Feifei Li, Yufei Tao, and Robert Christensen. Optimal Splitters for Temporal and Multi-version Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 109–120, New York, NY, USA, 2013.

[45] Susan Lyons. Persistent Identification of Electronic Documents and the Future of Footnotes. *Law Library Journal*, 97, 2005.

[46] Amélie Marian, Serge Abiteboul, Gregory Cobena, and Laurent Mignet. Change-Centric Management of Versions in an XML Warehouse. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 581–590, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[47] Hailey Mooney and Mark P. Newton. The Anatomy of a Data Citation: Discovery, Reuse, and Credit. *Journal of Librarianship and Scholarly Communication*, 1(1):6, 2012.

[48] Heiko Müller, Peter Buneman, and Ioannis Koltsidas. XArch: Archiving Scientific and Reference Data. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1295–1298, New York, NY, USA, 2008. ACM.

[49] Nature Publishing Group. Data's Shameful Neglect. *Nature*, 461(7261):145, 2009.

[50] OECD - Organisation for Economic Co-operation and Development. *Data and Metadata Reporting and Presentation Handbook*. OECD Publishing, Paris, 1. edition, 2007.

[51] Malay K. Pakhira. *Database Management System*. PHI Learning Pvt. Ltd., New Delhi, 2012.

[52] Norman Paskin. Digital Object Identifier (DOI) System. *Encyclopedia of Library and Information Sciences*, 3:1586–1592, 2008.

[53] Ronald Plew and Ryan Stephens. The Database Normalization Process, 2003. `http://www.informit.com/articles/article.aspx?p=30646`. Last access: 25-08-2015.

[54] Stefan Pröll and Andreas Rauber. Citable by Design A Model for Making Data in Dynamic Environments Citable. In *2nd International Conference on Data Management Technologies and Applications (DATA2013)*, July 2013.

[55] Stefan Pröll and Andreas Rauber. Scalable Data Citation in Dynamic Large Databases: Model and Reference Implementation. In *IEEE International Conference on Big Data 2013 (IEEE BigData 2013)*, October 2013.

[56] Ajay Rambhia. *XML Distributed Systems Design*. Sams Publishing, Indianapolis, Indiana, 2002.

[57] Andreas Rauber, Ari Asmi, Dieter van Uytvanck, and Stefan Pröll. Data Citation of Evolving Data, 2015. `https://rd-alliance.org/rda-wgdc-revised-version-recommendations-making-data-citeable-published.html`. Last access: 20-09-2015.

[58] Colin Ritchie. *Relational Database Principles*. Cengage Learning EMEA, Hampshire, 2002.

[59] Arif Shaon, Sarah Callaghan, Bryan N. Lawrence, Brian Matthews, Andrew Woolf, Timothy Osborn, and Colin Harpham. A Linked Data Approach to Publishing Complex Scientific Workflows. In *IEEE 7th International Conference on E-Science (e-Science), 2011*, pages 303–310, 2011.

[60] Martyn Shuttleworth. Reproducibility, 2009. `https://explorable.com/reproducibility`. Last access: 21-08-2015.

[61] Joan E. Sieber and Bruce E. Trumbo. (Not) Giving Credit Where Credit is due: Citation of Data Sets. *Science and Engineering Ethics*, 1(1):11–20, 1995.

[62] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering. In *Proceedings of the 16th International Conference on Data Engineering, 2000*, pages 547–558, 2000.

[63] Richard T. Snodgrass. Temporal Databases. *IEEE Computer*, 19:35–42, 1986.

[64] Richard T. Snodgrass. *Developing Time-oriented Database Applications in SQL*. Data Management Systems Series. Morgan Kaufmann Publishers, 2000.

[65] Kimbro Staken. Introduction to Native XML Databases, 2001. `http://www.xml.com/pub/a/2001/10/31/nativexmldb.html`. Last access: 07-02-2014.

[66] U.S. Department of the Interior | U.S. Geological Survey. Data Citation, 2015. http://www.usgs.gov/datamanagement/describe/citation.php. Last access: 24-08-2015.

[67] Mario Valle. Scientific Data Management, 2015. http://mariovalle.name/sdm/scientific-data-management.html. Last access: 16-09-2015.

[68] W3C. URIs, URLs, and URNs: Clarifications and Recommendations 1.0, 2001. www.w3.org/TR/uri-clarification. Last access: 01-06-2014.

[69] Priscilla Walmsley. *XQuery*. O'Reilly Media, Sebastopol, 2007.

[70] Fusheng Wang and Carlo Zaniolo. Temporal Queries in XML Document Archives and Web Warehouses. In *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic, 2003*, pages 47–55, July 2003.

[71] Jason Tsong-li Wang, Kaizhong Zhang, Karpjoo Jeong, and Dennis Shasha. A System for Approximate Tree Matching. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):559–571, August 1994.

[72] Jelte M. Wicherts and Marjan Bakker. Publish (Your Data) or (let the Data) Perish! Why not Publish Your Data too? *Intelligence*, 40(2):73 – 76, 2012.

[73] Yale Law School Roundtable on Data and Code Sharing. Reproducible Research. *Computing in Science Engineering*, 12(5):8–13, 2010.

[74] Shuohao Zhang and Curtis E. Dyreson. Adding Valid Time to Xpath. In *Proceedings of the Second International Workshop on Databases in Networked Information Systems*, pages 29–42, London, UK, 2002. Springer-Verlag.