UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4 Informatik

Persistente Speicherung von XML-Dokumenten in relationalen Datenbanken

Diplomarbeit

zur Erlangung des Grades einer Diplom-Informatikerin
im Studiengang Informatik

vorgelegt von
Katharina Ollinger
Matrikelnummer 200210422

Erstgutachter: Prof. Dr. Kurt Lautenbach,
Institut für Softwaretechnik, Fachbereich 4
Betreuer und Zweitgutachter: Dr. Stephan Philippi,
Institut für Softwaretechnik, Fachbereich 4

Koblenz, im Oktober 2006

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Inhaltsverzeichnis

1 Einleitung

XML durchdringt immer stärker alle Bereiche der Informationstechnologie, wenn es um die Darstellung und den Austausch von Daten und Dokumenten jeglicher Art geht. Dabei liegt der Vorteil von XML in der Einfachheit, durch die eine fachübergreifende Anwendung ermöglicht wird [See03].
Eine Konsequenz dieses Erfolgs ist die Notwendigkeit des Einsatzes effizienter Speichermethoden für XML. Es sollte möglich sein, Anfragen an die gespeicherten XML-Daten zu stellen und Änderungsoperationen durchzuführen.
Es gibt vielfältige Möglichkeiten, XML-Dokumente zu speichern. Ein Ansatz für die Speicherung ist, auf existierende Datenbank-Management-Systeme zurückzugreifen, die häufig auf dem relationalen Datenmodell basieren.
Im Rahmen des Projektpraktikums „XML-Datenbanken: Entwicklung einer Middleware“ an der Universität Koblenz wurde eine Java-basierte Middleware entworfen und implementiert, die es einem Client ermöglicht, XML-Dokumente beliebiger Struktur in einer relationalen Datenbank zu speichern (siehe Abbildung 1). Der Name der Middleware ist Jamrox „Java Middleware for Relational Storage of XML-Documents“[Pro04]. Diese erste Version der Middleware wird im Folgenden mit „Version nach [Pro04]“ oder als „erste Version“ bezeichnet. Dieser Prototyp wurde im Rahmen einer Studienarbeit weiterentwickelt und verbessert.
Diese Version, die den Abschluss der Studienarbeit darstellt, ist Ausgangspunkt dieser Arbeit. Sie wird von nun an als „vorliegende Version“ oder „Version nach [Sie05]“ bezeichnet.
Ziel dieser Arbeit ist, diese Version der Middleware vom Prototypen in Richtung eines auslieferbaren Produktes weiterzuentwickeln. Die Weiterentwicklung oder Wartung einer Software kann in unterschiedliche Richtungen gehen. So unterscheidet [Swa90] folgende Arten von Wartung:
Bei der korrektiven Wartung wird das System geändert, indem Fehler beseitigt werden.
Bei der adaptiven Wartung wird eine Änderung am System vorgenommen, um es an eine veränderte Systemumgebung anzupassen.
Die Verbesserung der Qualität eines Systems wird bei der perfektiven Wartung beabsichtigt. Hierbei ist die Verbesserung der Benutzer-Interaktion, der Effizienz und der Dokumentation von Interesse.
In dieser Arbeit soll das System hauptsächlich perfektiv weiterentwickelt werden. Korrektive Veränderungen werden vorgenommen, falls Fehler gefunden werden. Da es keine veränderte Umgebung gibt, in die die Middleware zu integrieren ist, muss keine adaptive Wartung stattfinden.
Bei der Weiterentwicklung sollen Schwerpunkte auf die Aspekte Funktionsumfang, Performance und Zuverlässigkeit gelegt werden.
Die veränderte Version der Middleware, die als Ergebnis dieser Arbeit entstehen soll, wird im Folgenden als „neue Version“ bezeichnet.
In den einzelnen Kapiteln dieser Arbeit sollen folgende Fragen beantwortet werden:

- Kapitel 2: wie ist die vorliegende Version der Middleware aufgebaut?
- Kapitel 3: was soll sich in der neuen Version der Middleware ändern?
- Kapitel 4: wie sollen die neuen Anforderungen umgesetzt werden?
- Kapitel 5: wie wurden die Anforderungen tatsächlich umgesetzt?
- Kapitel 6: wie hat sich die Performance in der neuen Version verbessert?

In Kapitel 7 sollen die Ergebnisse zusammengefasst und bewertet werden und noch offen gebliebene Fragen und Aufgaben dargestellt werden.
In Kapitel 2 - 5 orientieren sich die Unterkapitel jeweils an den eben vorgestellten Schwerpunkten für die Weiterentwicklung: Funktionsumfang, Performance und Zuverlässigkeit.

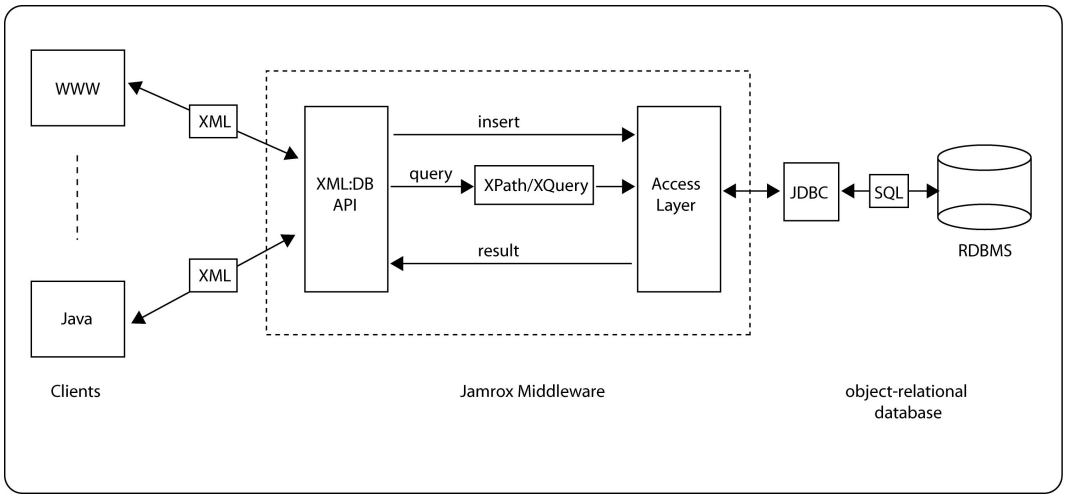


Abbildung 1: Architektur der Jamrox-Middleware

2 Beschreibung des Vorzustandes

In diesem Kapitel sollen die Eigenschaften und die Funktionsweise der Middleware in der Version nach [Sie05] näher beschrieben werden.

2.1 Funktionsweise und -umfang

2.1.1 Mapping-Schema

Um XML-Dokumente beliebiger Struktur in einer relationalen Datenbank speichern zu können, bedarf es eines Mapping-Schemas, welches das XML-Dokument auf eine Datenbank-Tabelle abbildet. Ein Mapping-Schema beschreibt, wie die verschiedenen Bestandteile eines XML-Dokuments (Elemente, Attribute, Text usw.) in der Datenbank gespeichert werden sollen.
Eine Auswahl verschiedener Mapping-Schemata wird in „A performance evaluation of alternative mapping schemes for storing xml in a relational database“ von Daniela Florescu und Donald Kossmann vorgestellt und verglichen [Kos99].
Bevor die erste Version der Middleware erstellt wurde, wurden verschiedene Mapping-Schemata bezüglich ihrer Performance verglichen. Das Schema „EdgeInline“ erwies sich in einer modifizierten Form als das effizienteste und wurde daher in der Middleware benutzt [Pro04].
Wie die Abbildung des XML-Dokuments auf die Datenbanktabelle mit dem erweiterten EdgeInline-Schema funktioniert, soll nun an einem Beispiel erläutert werden. Dazu wird die XML-Datei „DVD-Sammlung“(siehe Abbildung 2) in einer Edge-Inline-Tabelle (Abbildung 3) dargestellt.
Die *id*-Spalte dient der eindeutigen Identifizierung der Zeilen in der Datenbank und somit aller Bestandteile des XML-Dokuments.
In der *name*-Spalte steht der Name des Elements oder Attributs. Auf die Id des Elements der nächsthöheren Ebene wird in der *parent*-Spalte referenziert, so dass die Hierarchie eines XML-Dokuments rekonstruierbar ist. Handelt es sich um das Wurzelelement, ist dieser Wert auf -1 gesetzt.
Die Spalte *src* zeigt die Zugehörigkeit eines XML-Elements bzw. Attributs zu einem Objekt an, wobei XML-Elemente, die in der ersten Ebene unmittelbar nach dem root-Element stehen, als Objekte bezeichnet werden.
Um die richtige Reihenfolge der XML-Elemente einer Ebene zu erhalten, werden die XML-Elemente einer Ebene durchnummeriert und diese Nummer in der *ord*-Spalte festgehalten.
Um Attribute von Elementen unterscheiden zu können, wird in der *attribut*-Spalte angegeben, ob es sich um ein Attribut handelt oder nicht.
In der letzten Spalte *val* wird der Wert des jeweiligen Elements oder Attributs angegeben, sofern vorhanden. Dabei entspricht der Wert eines Elements dem Inhalt des Text-Kindknotens. Sind mehrere Text-Kindknoten vorhanden, wird nur das letzte als Wert gespeichert.

```
<?xml version="1.0"?>
<dvd-sammlung>
   <dvd laufzeit="114">
      <titel genre="Doku">Bowling for Columbine></titel>
      <regie>Michael Moore</regie>
   </dvd>
   <dvd laufzeit="181">
      <titel>The Green Mile</titel>
      <regie>Frank Darabont</regie>
   </dvd>
</dvd-sammlung>
```

Abbildung 2: XML-Datei „DVD-Sammlung“

id	*name*	*parent*	*src*	*ord*	*attribut*	*val*
1	dvd-sammlung	-1	1	1	FALSE	
2	dvd	1	2	1	FALSE	
3	laufzeit	2	2	1	TRUE	114
4	titel	2	2	1	FALSE	Bowling for Columbine
5	genre	4	2	1	TRUE	Doku
6	regie	2	2	2	FALSE	Michael Moore
7	dvd	1	7	2	FALSE	
8	laufzeit	2	7	1	TRUE	181
9	titel	2	7	1	FALSE	The Green Mile
11	regie	2	7	2	FALSE	Frank Darabont

Abbildung 3: DVD-Sammlung.xml im modifizierten EdgeInline-Schema

2.1.2 XML:DB-API

Um Entwicklern eine generalisierte Schnittstelle zur Datenbank zu bieten, wurde die XML:DB-API[1] ausgewählt. Durch diese wird die Konstruktion von Anwendungen ermöglicht, welche Daten in einer XML-Datenbank speichern, finden, anfragen und verändern wollen [XML06a]. Für diese Aktionen bietet die API entsprechende Konstrukte, die für die jeweilige Datenbank implementiert werden müssen.
So kann von der zugrunde liegenden Datenbank abstrahiert werden. Die API kann äquivalent zur JDBC[2] [Sun06d]-Technologie gesehen werden.
In Abbildung 4 ist die typische Architektur dargestellt, in der die XML:DB-API vorkommt. Verschiedene Anwendungen können die XML:DB-API benutzen und müssen dabei keine Rücksicht auf spezifische Anforderungen der darunter liegenden Datenbank nehmen. Für die verschiedenen Datenbanken muss es allerdings eine spezifische Implementierung der XML:DB-API geben, die mit den JDBC-Treibern vergleichbar ist.
So gibt es bereits eine XML:DB-API-Implementierung für die native XML-Datenbank eXist [eXi06b].

[1] Application Programmer Interface
[2] Java Database Connectivity

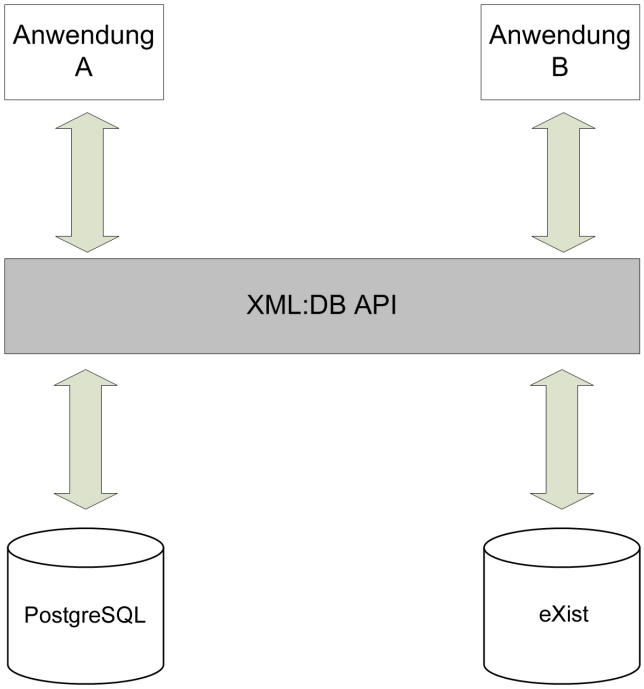


Abbildung 4: Architektur der XML:DB-API

Im Folgenden soll nun eine kurze Einführung in die Grundkonzepte der API gegeben werden. Nähere Informationen finden sich auf der Projekt-Website [XML06a].
Die API besteht aus Modulen, die in den API Core Level Specifications zusammengefasst sind (siehe Abbildung 5). So kann je nach Bedarf nur ein Teil der Core Level implementiert werden bzw. über die Core Level hinaus Funktionalität hinzugefügt werden.
XML-Dateien werden in der XML:DB-API als Resourcen bezeichnet, welche in einer Collection zusammengefasst werden. Eine Datenbank kann mehrere Collections enthalten, die in einer Baumstruktur gespeichert sind.
Die vorliegende Version der Middleware ist eine Implementierung der XML:DB-API. Die durch die API vorgegebene Struktur wurde in Jamrox durch folgende Relationen auf ein relationales Schema abgebildet [Pro04]:

- Die Relation „MetaCollectionTable“ enthält die Id's und Namen der gespeicherten Collections und für jede Collection die Id der Collection, die sich eine Hierarchieebene darüber befindet. Die Root-Collection hat für diesen Eintrag den Wert `null`.
- Die Relation „MetaDocumentTable“ beinhaltet die vorhandenen Resourcen mit Id, Name der XML-Datei und dem Tabellennamen.
- Die Relation „MetaConnectionTable“ stellt in Zweiertupeln dar, welche Resource zu welcher Collection gehört.

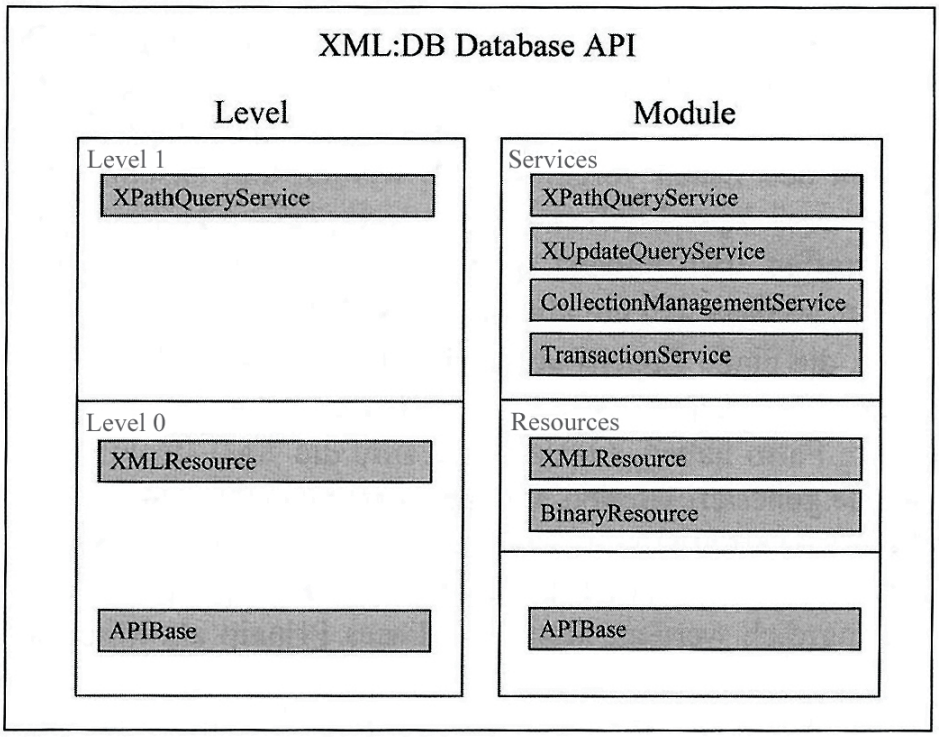


Abbildung 5: Aufbau der XML:DB-API [See03]

In der vorliegenden Version der Middleware Jamrox wurde das grundlegende Modul API Base vollständig und das Modul XMLResource unvollständig implementiert. Diese beiden Module sind für Core Level 0 notwendig.
Außerdem wurde Core Level 1 implementiert. Zu diesem gehört zusätzlich zu Core Level 0 der XPathQueryService. Auf diesen wird im nächsten Unterkapitel genauer eingegangen.
Grundlegend für das Arbeiten mit Collections ist der CollectionManagementService, mit dem Collections erzeugt und gelöscht werden können. Dieser wurde ebenfalls implementiert.
In der Implementierung der API erfolgt der Zugriff auf die Datenbank mit JDBC. Die Konstrukte der XML:DB-API zum Speichern, Löschen oder Abfragen von XML-Dokumenten werden mit JDBC-Statements umgesetzt, mit denen die jeweilige Änderung oder Anfrage auf der relationalen Datenbank durchgeführt wird. Die XML-Dokumente werden mit dem in Kapitel 2.1.1 beschriebenen Mapping-Schema in der relationalen Datenbank gespeichert.

2.1.3 XPathQueryService

Mit dem XPathQueryService wird es dem Benutzer ermöglicht, mittels XPath [W3C06c] Such-Anfragen an Resourcen in der Datenbank zu stellen. XPath ist eine Sprache, die für die Adressierung von Teilen eines XML-Dokuments entworfen wurde. Es werden zusätzlich Hilfsmittel angeboten, um Zeichenketten, boolsche Werte und Zahlen zu manipulieren. Durch die Verwendung einer auch in URL's genutzten Pfad-Notation erhält XPath seinen Namen. Mit diesem „path“ lässt sich durch die hierarchische Struktur eines XML-Dokuments

navigieren [W3C06d].
Es würde im Rahmen dieser Arbeit zu weit führen genauer auf XPath einzugehen. Daher soll hier nur auf die kommentierte, deutsche Übersetzung der W3C-Spezifikation [W3C06d] verwiesen werden.
In der vorliegenden Version der Middleware wurde der XPathQueryService mit Hilfe der Open Source XPath Engine Jaxen [Cod06b] implementiert.
Jaxen ist eine universelle Java XPath Engine, die XPath 1.0-Ausdrücke parst und gegen beliebige XML-Objektmodelle auswertet. Jaxen ist für kommerzielle Nutzung frei verfügbar und kann an verschiedene Objektmodelle angepasst werden [Cod06b]. In der Version nach [Sie05] wurde für den XPathQueryService das DOM[3] gewählt.
DOM ist ein W3C-Standard, der verschiedene Möglichkeiten festlegt, wie auf ein Dokument zugegriffen werden kann. DOM legt diese Eigenschaften für alle strukturierten Inhaltsmodelle fest, so auch für XML. Der Vorteil vom DOM liegt darin, dass es programmiersprachenunabhängig ist und einen einfachen Zugriff auf die Inhalte des Dokuments über entsprechende Objekte ermöglicht [Nie04].
DOM stellt die Datenstruktur eines XML-Dokuments durch einen Baum von Knoten dar. In Java erfolgt der Zugriff auf den DOM-Baum über das Paket `org.w3c.dom [Sun06a]`. Verschiedene Interfaces in diesem Paket repräsentieren die unterschiedlichen Typen von Knoten (z.B. Element, Attribut, Kommentar). Dabei sind alle Interfaces vom allgemeinen Java-Interface `Node` abgeleitet, welches grundlegende Methoden zur Navigation und zum Zugriff auf den Baum bereitstellt [Har03].
In der Middleware wurde ein eigenes DOM-Objektmodell implementiert, das die relationale Datenbank als DOM-Baum darstellt. Das heißt, es wurde für jeden DOM-Knoten (z.B. das DOM-Element) der zugehörige Jamrox-Knoten definiert, der die Informationen eines Datenbank-Tupels in Form eines DOM-Knotens darstellt.
Diese spezifische Implementierung der DOM-Knoten wurde in der Middleware im Paket `org.jaxen.jamrox` zusammengefasst und enthält beispielsweise die Klassen `JamroxElement`, `JamroxAttr` und `JamroxText`.
In der Implementierung der Methoden der verschiedenen Knotentypen wird auf die Datenbank zugegriffen, um Informationen über den jeweiligen Knoten bereitzustellen. In der Methode `getNextSibling()` in der Klasse `JamroxElement` wird z.B. mit folgendem SQL-Statement der Geschwisterknoten des Elements „aktElement“ gesucht:

```
SELECT * FROM tabelle
WHERE parent = "(aktElement:parent)"
AND ord = "(aktElement:ord+1)"
```

Dadurch wird es der Jaxen XPath Engine ermöglicht, auf dem XML-Dokument, also auf der Datenbank, zu navigieren.

[3]Document Object Model

2.2 Performance

Um die Performance der Middleware bewerten zu können, wurden Messungen durchgeführt, welche die Version nach [Sie05] mit der ersten Version der Middleware und der nativen Open-Source XML-Datenbank eXist [eXi06b] vergleichen.
Hierzu wurde die Zeit für das Einlesen von XML-Dokumenten und Bearbeiten von 8 verschiedenen XPath-Anfragen gemessen [Sie05]. Dabei wurde festgestellt, dass die vorliegende Version langsamer arbeitet als die erste Version, was aber an der unvollständigen und fehlerhaften Implementierung der ersten Version liegt.
Außerdem wurde bei den Messungen herausgefunden, dass bei der vorliegenden Version die Bearbeitung der XPath-Anfragen länger dauert als mit eXist. Die Geschwindigkeit der Middleware ist stark von der Performance der Datenbank abhängig. Optimierungen wurden bereits erreicht durch Anlegen eines Indexes auf der *parent*-Spalte und durch Umstellung der SQL-Statements auf „prepared Statements“.

2.3 Zuverlässigkeit

Die Zuverlässigkeit der Middleware wurde mit JUnit-Tests [Bec06] gesichert. JUnit ist ein Open Source Regression Testing-Framework, mit dem einzelne Java-Programmteile isoliert auf funktionale Korrektheit getestet werden können [Bec06].
Es werden alle XML:DB-API-Methoden und die Jamrox-DOM-Implementierung getestet. Für Letzteres wurde die Testklasse `JamroxDOMTest` erstellt, in der die Navigation auf dem Jamrox-DOM-Objektmodell getestet wird.
Beim Testen der XML:DB-API-Implementierung wurde besonderen Wert darauf gelegt, dass die Implementierung der Middleware der Spezifikation der XML:DB-API entspricht.
Die JUnit-Tests sind unabhängig von der spezifischen Implementierung der Middleware, so dass die geforderten Funktionalitäten ausschließlich mit den von der XML:DB-API bereitgestellten Methoden überprüft werden. So bleiben die Tests unabhängig von der Implementierung und können bei einer Änderung dieser bestehen bleiben.

3 Anforderungen für die Weiterentwicklung

Für die im letzten Kapitel vorgestellte Middleware bestehen verschiedene Weiterentwicklungsmöglichkeiten. Im Folgenden soll erklärt werden, welche Anforderungen an die neue Version gestellt werden.

3.1 Funktionsumfang

3.1.1 DB-Client

Wie in Abbildung 1 dargestellt, ist es möglich, verschiedene Clients an die Jamrox-Middleware anzubinden. Es existiert ein DB-Client „Titanium“ [Tit06], der dem Benutzer eine grafische Oberfläche für XML-Datenbanken bietet, welche die XML:DB-API implementieren. Diese XML:DB-GUI ermöglicht dem Benutzer folgende Aktionen [XML06d]:

- Navigation im Collection-Tree
- Einfügen, Löschen und Verändern von Resourcen
- Hinzufügen oder Entfernen von Collections
- Importieren oder Exportieren von Dokumenten (aus dem bzw. in das Dateisystem)
- Suche mit XPath-Ausdrücken (mit Unterstützung von Namensräumen)

In den vorhergehenden Arbeiten wurde bereits ohne Erfolg versucht, den Client an die Middleware anzubinden. Im Rahmen dieser Arbeit soll dieser Client nun für die Middleware nutzbar gemacht werden.

3.1.2 Unterstützung von SAX und DOM

SAX und DOM bieten zwei unterschiedliche Möglichkeiten, XML-Daten darzustellen und zu verarbeiten. Auf die Funktionsweise von SAX wird in Kapitel 5.1.1 genauer eingegangen. DOM wurde bereits in Kapitel 2.1.3 beschrieben. In der vorliegenden Version fehlt die Funktionaliät, die den Inhalt einer Resource in Form von SAX-Events liefert. Es sollte auch möglich sein, den Inhalt einer Resource auf den Inhalt eines DOM-Knotens bzw. über SAX-Events zu setzen. Diese Funktionalitäten sollen in der neuen Version enthalten sein.

3.1.3 Unterstützung von Namensräumen

Namensräume sind ein wichtiges Konzept innerhalb der XML-Spezifikation. Es beschreibt die Bindung zwischen den folgenden drei Teilen [Nie04]:

- eine eindeutige URI (z.B. `http://foo.bar`)
- ein Präfix (z.B. `foo:`)
- ein XML-Element oder Attribut (z.B. `<person/>`)

Um ein XML-Element weltweit eindeutig zu machen, kann ein Präfix an eine URI gebunden und dieses Präfix wiederrum dem XML-Element zugeordnet werden. Die Eindeutigkeit wird dadurch erreicht, dass die URI in der Regel dem Domainnamen der Entwicklergruppe entspricht [Nie04].
Die Deklarierung des Namensraums kann z.B. so aussehen:

```
<person xmlns:foo="http://foo.bar"/>
```

Dadurch kann das Element, in dem der Namensraum deklariert wurde, und/oder beliebige Nachkommen an die URI und damit an den Namensraum gebunden werden, z.B. so:

```
<foo:person xmlns:foo="http://foo.bar">
   <name>Markus Muster</name>
</foo:person>
```

Das `<person/>`-Element hat damit den lokalen Namen `person`, das Präfix `foo`, den qualifizierten Namen `foo:person` und die Namensraum-URI `http://foo.bar`. Das Element `<name/>` ist keinem Namensraum zugeordnet. Attribute können auf die gleiche Weise an einen Namensraum gebunden werden.
Um nicht jedes Mal das Präfix vor den Elementnamen schreiben zu müssen, kann man Default-Namensräume definieren. Um z.B. das Element `<person/>` und alle Nachkommen an einen bestimmten Namensraum zu binden, wäre folgende Deklaration notwendig:

```
<person xmlns="http://foo.bar">
   <name>Markus Muster</name>
</person>
```

Durch das Weglassen des Präfixes bei der Deklaration wird der Default-Namensraum definiert und dadurch das Element `<person/>` und alle Nachkommen, die nicht explizit durch ein Präfix an einen anderen Namensraum gebunden sind, dem Default-Namensraum zugeordnet [Nie04].
Damit wären beide Elemente (`<person/>` und `<name/>`) an den Namensraum `http://foo.bar` gebunden. Attribute können nicht in einem Default-Namensraum liegen.
Jamrox speichert bisher keine Namensraumdeklarationen und es werden nur lokale Namen von Elementen und Attributen in der Datenbank abgelegt. Dies soll geändert werden.
Durch die fehlende Unterstützung von Namensräumen kann im XPathQueryService bisher keine Anfrage durchgeführt werden, in der Elemente oder Attribute enthalten sind, die in Namensräumen liegen. Außerdem konnte die Jaxen-Klasse `JamroxNamespaceNode` nicht benutzt werden, die für die Nutzung der Namensraum-Achse in XPath notwendig ist. Die Namensraum-Achse liefert alle Namensraumdeklarationen einer ausgewählten Knotenmenge.
Das Ignorieren von Namensräumen in der aktuellen Version der Middleware hat außerdem zur Folge, dass folgende Node Set Functions in XPath nicht genutzt werden können:

- `string local-name(node-set?)`
- `string namespace-uri(node-set?)`
- `name(node-set?)`

Diese Funktionen liefern den lokalen Namen, den Namensraum oder den qualifizierten Namen eines Knotens. Ihre Benutzung soll in der neuen Version der Middleware möglich sein.

3.1.4 Unterstützung von CDATA-Abschnitten, Kommentaren, Entity-Referenzen und Processing-Instructions

Außer XML-Elementen und Attributen gibt es noch folgende Konstrukte, die in einem XML-Dokument vorkommen können: Entity-Referenzen, CDATA-Abschnitte, Kommentare und Processing-Instructions.
Diese Konstrukte sollen im Folgenden vorgestellt werden:

- Entity-Referenzen werden benutzt, um bestimmte Zeichen im Text darzustellen, die bereits reserviert sind und eine spezielle Bedeutung haben [Nie04].
  So würde z.B. der Text `10 < 20` in einem XML-Element folgendermaßen dargestellt, weil das Zeichen „<“ in XML bereits reserviert ist:

  ```
  <term>10 &lt; 20</term>
  ```

- Um nicht für alle reservierten Zeichen Entity-Referenzen benutzen zu müssen, kann Text, der vom Parser nicht interpretiert werden soll, in einen CDATA-Abschnitt geschrieben werden. Das sieht z.B. so aus:

  ```
  <term><![CDATA[10 < 20]]</term>
  ```

  Dies ist vor allem für Texte sinnvoll, die mehrere reservierte Zeichen enthalten.

- Kommentare beginnen in XML mit `<!--` und enden mit `-->`.

- Processing-Instructions geben bestimmten Anwendungen Informationen darüber, wie das Dokument behandelt werden soll, nachdem es geparst wurde. Ein Beispiel ist das `xml:stylesheet`, das einem Browser mitteilt, wo sich das Stylesheet befindet, welches auf das Dokument angewendet werden soll [Har04].
  Processing-Instructions beginnen mit `<?`, gefolgt von einem Namen, der das Ziel der Processing-Instruction darstellt, gefolgt von einem Leerzeichen und beliebigen Zeichen, gefolgt von `?>`.

Die gerade vorgestellten XML-Konstrukte werden in der aktuellen Version nicht in der Datenbank gespeichert, sondern ignoriert. Das hat zur Folge, dass ein XML-Dokument unter Umständen nicht vollständig aus der Datenbank rekonstruiert werden kann.
Ziel soll es sein, die fehlende Unterstützung in der Middleware zu ergänzen

und so die exakte Rekonstruktion der Dokumente zu ermöglichen.
Durch die fehlende Unterstützung der oben genannten Konstrukte konnten die XPath-Node Tests `comment()` und `processing-instruction()` im XPathQueryService nicht benutzt werden. Diese Knotentests sollen nach der Ergänzung der Konstrukte ebenfalls benutzbar sein.

3.1.5 Speichern von Elementen mit gemischtem Inhalt

Bisher ist es mit der Middleware nicht möglich, XML-Elemente mit gemischtem Inhalt, das heißt mit mehreren Text-Kindknoten, die durch andere XML-Knoten getrennt werden, zu speichern.
Jedes XML-Element darf höchstens **einen** Textknoten als Kind haben, dessen Inhalt ihm als Wert (in der Spalte *val*) in der DB-Tabelle zugeordnet wird. Zusätzliche Kind-Textknoten gehen bei der Abbildung auf das EdgeInline-Schema verloren. Ein Beispiel zur Verdeutlichung folgt:

```
<ARTIST>
   not found
   <NAME>Bob</NAME>
   not found
   <LASTNAME>Dylan</LASTNAME>
   found
<ARTIST>
```

Bei diesem XML-Element würden die zwei `not found`-Textknoten verloren gehen. Nur der Textknoten `found` würde als Wert gespeichert. In der neuen Version der Middleware soll es möglich sein, auch mehrere Text-Kindknoten zu speichern.

3.1.6 Einbau des TransactionServices

Außer dem XPathQueryService gibt es noch weitere Services, deren Interfaces implementiert werden können (siehe dazu Abbildung 5). Es gibt einen Service, der es ermöglichen soll, mehrere Veränderungen an einer Collection in einer Transaktion zusammenzufassen [XML06c].
Dieser TransactionService soll in der neuen Version der Middleware implementiert und integriert werden.

3.1.7 Einbau des XUpdateQueryServices

Bisher können nur lesende Anfragen an die Datenbank gestellt werden. In der neuen Version soll nun auch der XUpdateQueryService implementiert werden, mit dem Updates mit der XML:DB-Update-Sprache auf Collections und Resourcen ausgeführt werden können [XML06h].

3.1.8 Validierung

Eine Stärke von XML ist, dass die Struktur des Inhalts völlig frei bestimmt werden kann. Für manche Anwendungen möchte man genau diese Freiheit aber beschränken, indem man Bedingungen aufstellt, wie ein XML-Dokument

aufgebaut sein muss und diese in einer DTD [4] oder in einem XML Schema festhält [Nie04]. In dem jeweiligen XML-Dokument muss dann ein Verweis auf die DTD oder das XML Schema enthalten sein, so dass der Parser überprüfen kann, ob das Dokument die Bedingungen auch wirklich einhält. Diesen Vorgang des Prüfens nennt man „Validierung“. Schließt sie mit einem positiven Ergebnis ab, bezeichnet man das Dokument als gültig.
Um die Gültigkeit eines Dokuments nach einem Update zu erhalten, soll in der Middleware jedes Update auf Integritätsbedingungen, die in der zugehörigen DTD oder dem XML Schema festgehalten sind, überprüft werden.
Erst wenn festgestellt wurde, dass dieses Update gegenüber der DTD oder dem Schema gültig ist, soll das Update durchgeführt werden.

3.2 Performance

Bei den Perfomance-Messungen wurde festgestellt, dass die Performance der Middleware hinter der von eXist zurückbleibt.
Es sollen verschiedene Möglichkeiten zur Steigerung der Performance beim Einfügen von Dokumenten und bei Anfragen untersucht und realisiert werden.

3.3 Zuverlässigkeit

Die Implementierung der API und die Darstellung der Datenbank als DOM-Baum wurden in den vorangegangenen Arbeiten bereits mit JUnit getestet. Diese Testfälle sollen überarbeitet und an die neuen Funktionalitäten angepasst und gegebenenfalls erweitert werden.
Bisher wurde der Quellcode noch nicht auf Coding-Standards und Bugs untersucht, was im Rahmen dieser Arbeit nachgeholt werden soll.

[4] Document Type Definition

4 Design der neuen Version

Dieses Kapitel stellt nun dar, wie die Forderungen des letzten Kapitels bezüglich Funktionalität, Performance und Zuverlässigkeit in der neuen Version der Middleware umgesetzt werden sollen.

4.1 Funktionsumfang

4.1.1 Anbindung an XML:DB-GUI

Damit die Middleware mit Hilfe einer grafischen Benutzungsoberfläche bedient werden kann, soll der in Kapitel 3.1.1 erwähnte Titanium-Client an die Jamrox-Middleware angebunden werden.
Hierzu sind eine spezielle Konfiguration des Clients und eventuell zusätzliche Anpassungen in der Middleware notwendig.

4.1.2 Veränderung des EdgeInline-Schemas

Das Mapping-Schema muss verändert werden, um die in Kapitel 3.1.4 angesprochene Unterstützung der CDATA-Abschnitte, Processing-Instructions, Kommentare und Entity-Referenzen zu ermöglichen, damit Elemente mit gemischtem Inhalt gespeichert werden (Kapitel 3.1.5) und um Namensräume zu unterstützen (Kapitel 3.1.3).
Dazu soll sich das neue Mapping-Schema am Document Object Model (DOM) orientieren, da dieses auch für den XPathQueryService benutzt wird. DOM unterscheidet u.a. zwischen den folgenden Knotentypen [Har03]:

- `Attribute`
- `CDATA-Section`
- `Comment`
- `Document`
- `Element`
- `Entity Reference Node`
- `Processing-Instruction`
- `Text`

Um das XML-Dokument wieder exakt aus der Datenbank reproduzieren zu können, sollte diese Unterscheidung auch in das EdgeInline-Schema übernommen werden.
Deswegen soll die Attribut-Spalte durch eine „Knotentyp“-Spalte ersetzt werden, deren Wert einem der eben aufgezählten Typen entspricht.
Die Knotentypen `Document_Fragment_Node`, `Entity` und `Notation` gibt es ebenfalls im DOM, sie wurden in dieser Aufzählung nicht berücksichtigt, da sie für unsere Zwecke nicht von Belang sind. Sie sind nicht Teil des DOM-Baums und werden nur für spezielle Zwecke gebraucht.

Außerdem wurde der Typ `Node` nicht aufgezählt, weil im DOM alle Knoten von diesem Typ sind und dieser Typ deswegen nicht als Merkmal für unsere Unterscheidung zu gebrauchen ist.
Besonders zu beachten ist, dass wir den Text-Knoten nun als eigenständigen Knoten betrachten, der eine eigene Zeile in der DB-Tabelle bekommt und nicht nur als Wert in der Spalte *val* in dem dazugehörigen Element-Knoten eingetragen wird.
So ist es möglich, Elemente mit gemischtem Inhalt (mehreren Text-Knoten und weiteren XML-Knoten) zu speichern, da nun einem Elementknoten nicht mehr nur der Wert **eines** Textknotens zugeordnet werden kann.
Der Wert eines Textknotens wird in der *val*-Spalte gespeichert, in der *parent*-Spalte wird auf die Id des Element-Vaterknotens referenziert und mittels der *ord*-Spalte wird der Textknoten in die Reihenfolge der Kindelemente eingeordnet.
Besitzt ein Knoten keinen Namen (wie z.B. Textknoten), so wird für dieses Tupel in der *name*-Spalte kein Wert eingetragen.
CDATA-Abschnitte und Kommentare werden ebenfalls ohne Namen und mit ihrem Text als Wert in der Tabelle abgelegt.
Bei Processing-Instructions wird das Ziel (z.B. xml-stylesheet) in der Spalte *name* und der Wert (alles was nach dem Ziel steht) in der Spalte *val* gespeichert. Steht eine Processing-Instruction noch vor dem Wurzelelement, ist ihre *parent*-Id ebenfalls -1.
Bei Entity-Referenzen ist zu beachten, dass sie aus dem umgebenden Text (falls vorhanden) extrahiert werden und als Geschwisterknoten zum vorhergehenden bzw. nachfolgenden Textknoten gespeichert werden. Dabei wird unter *name* die Entity-Referenz gespeichert und unter *val* das entsprechende Zeichen.
Um zukünftig Namensräume in der Middleware zu unterstützen, müssen zwei Maßnahmen getroffen werden:

1. Namespace-Deklarationen werden als Attribute in der Datenbank gespeichert. Dabei dient DOM wieder als Orientierung, bei dem es keine Unterscheidung zwischen Attribut- und Namensraumdeklarationen gibt.

2. Bei Elementen und Attributen wird in der Spalte *name* nicht mehr nur der lokale Name gespeichert, sondern der qualifizierte Name.

Durch die Speicherung der Namensraumdeklarationen und des Präfixes als Bestandteil des qualifizierten Namens ist es möglich, die URI eines Namensraumes zu ermitteln, die einem Element oder Attribut zugeordnet ist. Man muss dafür die Namensraumdeklaration suchen, in der das entsprechende Präfix auf die URI abgebildet wird. Dabei gibt es jedoch zwei Probleme:

1. Soll zu einem Element oder Attribut die zugehörige URI ausgegeben werden, müsste die ganze DB-Tabelle nach der Namensraumdeklaration durchsucht werden, in der das Präfix des Elements oder Attributs einer URI zugeordnet wird.
   Jaxen vergleicht bei XPath-Anfragen, welche Elemente oder Attribute enthalten, die in einem Namensraum liegen, nur die URIs, nicht die Präfixe. Das heißt, dass der Zugriff auf eine URI oft stattfindet, was bei einer

Vorgehensweise wie oben beschrieben zu Performance-Einbußen im XPathQueryService führen würde.

2. Mit XML ist es auch möglich, Default-Namensräume in einem Element zu deklarieren, was bedeutet, dass alle Nachfahren-Elemente (alle Kinder, Kindeskinder usw.) ohne Präfix diesem Namensraum zugeordnet sind. Attribute können nicht in einem Default-Namensraum liegen.
   Das bedeutet für die Datenbank, dass bei Betrachten eines Tupels vom Typ `Element` ohne Präfix im Namen nicht direkt ersichtlich ist, ob es einem Namensraum zugeordnet ist oder nicht. Bei solchen Elementen müsste man zuerst alle Vorfahren-Elemente anschauen, um herauszufinden, ob in einem Vorfahren-Element ein Default-Namensraum deklariert wurde.

Zusammenfassend kann man sagen, dass die nötigen Informationen, um die URI zu einem Element herauszufinden, alle in der Datenbank vorhanden, jedoch nur mit aufwändigen Datenbankzugriffen abrufbar sind.
Die URI in der Datenbank in einer zusätzlichen Spalte zu speichern, wäre zwar eine Lösung der beiden Probleme, würde aber dazu führen, dass redundante Informationen in der Datenbank gespeichert würden.
Deswegen soll folgende Lösung in der Middleware umgesetzt werden: Die Namensraumdeklarationen sollen zusätzlich zur Speicherung in der Datenbank in einem Java-Objekt gespeichert werden, welches jedem `JamroxDocument`, also jeder Resource, zugeordnet ist. So ist in der Implementierung der API ein schneller Zugriff auf eine URI möglich.
Zur Lösung des zweiten Problems soll jedem Default-Namensraum ein „konstruiertes“ Präfix zugeordnet werden. Jedem Element, das in diesem Default-Namensraum liegt, wird dieses Präfix in der Spalte *name* in der Datenbank wie ein normales Präfix vorangestellt.
Um das „konstruierte“ Präfix von einem echten Präfix zu unterscheiden, soll es mit dem Sonderzeichen „#“ beginnen, das bei normalen Präfixen nicht erlaubt ist. Um die Default-Namensräume voneinander zu unterscheiden, wird hinter das Raute-Zeichen noch eine Zahl gesetzt, d.h. die Präfixe werden einfach durchnummeriert.
Die letzte Änderung am Mapping-Schema ist das Entfernen der Spalte *src*. Diese Spalte speichert für jeden Knoten die Id desjenigen Vorfahren-Elements, das in der ersten Ebene des Dokuments liegt. Sie wird von nur zwei DOM-Methoden in den JDBC-Statements benutzt, wovon nur eine Methode bei der Achsennavigation in Jaxen genutzt wird.
Um die Änderungen am Mapping-Schema zu veranschaulichen, soll die Datei „rezept.xml“ (Abbildung 6) mit dem neuen Mapping-Schema auf die Datenbank abgebildet werden. Die dadurch entstandene Tabelle wird in Abbildung 7 dargestellt.
Anhand dieses Beispiels soll das Vorgehen bei der Speicherung nach dem neuen Schema nochmal verdeutlicht werden: Zuerst wird die Processing-Instruction mit *name* `xml-stylesheet` und *val* `href="style.xsl" type="text/xml"` in der DB gespeichert. Die *ord*-Id ist bei diesem Tupel 1 und die *parent*-Id -1, da die Processing-Instruction noch vor dem root-Element steht.

```
<xml version="1.0">
<?xml-stylesheet href="style.xsl" type="text/xml"?>
<rezept>
   <zt:zutat id="mehl"
      xmlns:zt="http://www.kochen.de/zutaten">
      200g Mehl
   </zt:zutat>
   <!-- weitere Zutaten -->
   <anleitung xmlns="http://www.kochen.de/anleitung">
      Zuerst nehmen Sie das
      <zutat>Mehl</zutat>
      und mischen es mit Zucker &amp; Eiern
   </anleitung>
   <![CDATA[Alles gut "durchschlagen"!]]>
</rezept>
```

Abbildung 6: XML-Datei „rezept.xml“

Das nun folgende root-Element `rezept` besitzt dieselbe *parent*-Id, hat aber eine *ord*-Id von 2.
Das Element `zt:zutat` wird mit seinem qualifizierten Namen in der DB abgelegt.
Der in diesem Element deklarierte Namensraum `xmlns:zt= "http://www.kochen.de/zutaten"` wird als Attribut (*type*=“Attribute“) und der Kindknoten `200g Mehl` als Text (*type*=“Text“) gespeichert.
Der nun folgende Kommentar `<!--weitere Zutaten-->` wird als Geschwisterknoten zu `<zt:zutat/>` auf ein DB-Tupel abgebildet.
In dem darauf folgenden Element `<anleitung/>` wird ein Default-Namensraum deklariert. Diese Deklaration ist in der DB-Tabelle im Tupel mit der *id* 9 dargestellt und bekommt das konstruierte Präfix „#1“ zugeordnet. Den in diesem Namensraum liegenden Elementen `<anleitung/>` und `<zutat/>` wird dieses Präfix in der *name*-Spalte als Präfix vorangestellt.
Das Element `anleitung` hat 5 Kindknoten: Der Erste ist der Textknoten `Zuerst nehmen Sie das` (Tupel mit *id* 10), dann folgt das Element `<zutat/>` (*id*=11), dann wieder ein Textknoten `und mischen es mit Zucker` (*id*=13), dann die Entity-Referenz `&` (*id*=14) und als Letztes der Textknoten `Eiern` (*id*=15).
Das letzte Tupel in der DB-Tabelle stellt den CDATA-Abschnitt `Alles gut "durchschlagen"!` dar, welcher der vierte Kindknoten (*ord*=4) des root-Elements ist (*parent*=2).
Die Änderung des EdgeInline-Schemas hat zur Folge, dass das Einlesen von XML-Dokumenten in der Klasse `EdgeInline.java` verändert werden muss.
Anpassungen an das neue Schema müssen im Paket `org.jaxen.jamrox` vorgenommen werden, da hier auf die DB-Tabelle zugegriffen wird, um sie als DOM darzustellen. In diesem Paket müssen die „neuen“ Knoten als Jamrox-Klassen hinzugefügt werden. Außerdem müssen Methoden, die Namensräume betreffen (z.B. getNamespaceURI()), implementiert werden.

id	name	parent	ord	type	val
1	xml-stylesheet	-1	1	P_I	href=“style.xsl“ type=“text/xml“
2	rezept	-1	2	Element	
3	zt:zutat	2	1	Element	
4	id	3	1	Attribute	mehl
5	xmlns:zt	3	2	Attribute	http://www.kochen.de/zutaten
6		3	1	Text	200g Mehl
7		2	2	Comment	weitere Zutaten
8	#1:anleitung	2	3	Element	
9	xmlns:#1	8	1	Attribute	http://www.kochen.de/anleitung
10		8	1	Text	Zuerst nehmen Sie das
11	#1:zutat	8	2	Element	
12		11	1	Text	Mehl
13		8	3	Text	und mischen es mit Zucker
14	amp;	8	4	E_R	&
15		8	5	Text	Eiern
16		2	4	CDATA	Alles gut „durchschlagen“!

Abbildung 7: „rezept.xml“ im neuen EdgeInline-Schema

4.1.3 Vollständige Implementierung der XML:DB-API

In der neuen Version der Middleware soll die XML:DB-API vollständig implementiert sein. Das heißt, dass auch DOM und SAX unterstützt werden (siehe Kapitel 3.1.2).
In der Klasse `XMLResourceImpl` werden dazu folgende Methoden implementiert:

- `getContentAsDOM()`
- `setContentAsDOM(Node node)`
- `getContentAsSAX(ContentHandler handler)`
- `setContentAsSAX()`

Da Namensräume nun in der Datenbank berücksichtigt werden (siehe 4.1.2), kann auch die Implementierung folgender Methoden in der Klasse `XPathQueryServiceImpl` ergänzt werden:

- `setNamespace(String URI, String prefix)`
- `getNamespace(String prefix)`
- `removeNamespace(String prefix)`
- `clearNamespaces()`

Diese Methoden erlauben dem XPathQueryService, XPath-Anfragen zu bearbeiten, in denen Elemente oder Attribute enthalten sind, die in einem Namensraum liegen. Erläuterungen zum Zweck dieser Methoden folgen in Kapitel

5.1.3).
Der TransactionService und der XUpdateQueryService werden ebenfalls umgesetzt (siehe Kapitel 3.1.6 und 3.1.7).
Für den TransactionService wird das Interface `TransactionService` mit den typischen Transaktionsmethoden `begin()`, `commit()` und `rollback()` implementiert.
Um den XUpdateQueryService bereitzustellen, muss das dazugehörige Interface mit seinen Methoden `update(String commands)` und `updateResource (String id, String commands)` implementiert werden.
Darüber hinaus soll, wie in Kapitel 3.1.8 gefordert, bei einem Update geprüft werden, ob das Dokument durch die Änderung gegenüber der dazugehörigen DTD oder dem Schema (falls vorhanden) gültig bleibt. Nur wenn dies der Fall ist, soll die Änderung in die Datenbank übernommen werden. Anderenfalls soll mit einer Fehlermeldung auf die Ungültigkeit des Updates hingewiesen werden.
Zur Realisierung dieser Funktionalität besteht die Möglichkeit, die Überprüfung auf Gültigkeit mit dem DOM-Parser durchzuführen. Dieser muss dazu allerdings das ganze Dokument einlesen, um festzustellen, ob das Dokument gültig ist oder nicht.
Eine Überprüfung dieser Art nach jedem Update kostet viel Zeit. Besonders bei großen Dokumenten ist diese Vorgehensweise daher nicht vertretbar.
Es kommt deshalb nur eine Lösung in Frage, bei der das Dokument inkrementell gegen die DTD oder das Schema überprüft wird, d.h. dass nicht das ganze Dokument betrachtet werden muss, sondern nur der Teil, der verändert werden soll.
Dafür gibt es in DOM die Spezifikation Document Object Model Validation Level 3 [W3C06a], die plattform- und sprachenneutral eine Schnittstelle beschreibt, mit der das dynamische Überprüfen von Gültigkeit ermöglicht werden soll.
Die Java-Sprachbindung ist im Paket `org.w3c.dom.validation` zusammengefasst. Hier gibt es z.B. ein Interface `NodeEditVAL`, das Methoden wie `canInsert-Before(Node newChild, Node refChild)` oder `canAppendChild(Node newChild)` bereitstellt.
Diese Methoden würden für jedes mögliche Update-Konstrukt eine entsprechende Überprüfung auf Wohlgeformtheit erlauben.
Der Einsatz dieses Pakets entspricht folglich genau den Vorstellungen von einer inkrementellen Überprüfung und soll deshalb in die Middleware integriert werden.

4.2 Performance

4.2.1 Steigerung der Geschwindigkeit beim Einfügen von Dokumenten

Das Einfügen von Dokumenten könnte durch eine veränderte Einstellung des JDBC-Treibers beschleunigt werden. Bisher blieb die Autocommit-Funktion eingeschaltet, was bedeutet, dass nach jeder Anweisung die Änderung automatisch festgeschrieben wird.
Nachdem in der Middleware der TransactionService (siehe Kapitel 3.1.6) inte-

griert wurde, kann der AutoCommit-Modus beim Einfügen eines Dokuments deaktiviert werden. So kann erst nach der Erzeugung der Statements zum Einfügen des Dokuments der „Commit“-Befehl durchgeführt werden.

4.2.2 Steigerung der Geschwindigkeit bei Anfragen

Die Bearbeitung von XPath-Queries im XPathQueryService soll beschleunigt werden. Verschiedene Ansatzpunkte zum Erreichen dieses Ziels sollen im Folgenden vorgestellt werden:

- XPathQuery Rewriting: es soll geprüft werden, ob es für XPath-Ausdrücke eine „Normalform“ gibt.
  Ist dies der Fall, soll untersucht werden, ob durch das Umformen einer XPath-Anfrage in die „Normalform“ eine Performance-Steigerung beim Bearbeiten von XPath-Anfragen möglich ist.

- Optimierung von SQL-Anfragen bei der Achsennavigation: grundlegend für die Achsennavigation in Jaxen sind die SQL-Anfragen, welche die Informationen zu einer Achse suchen und bereitstellen.
  Es soll untersucht werden, ob die SQL-Statements, mit denen der Zugriff auf die Datenbank erfolgt, noch weiter vereinfacht werden können. Dadurch könnte sich eine Beschleunigung des Datenbankzugriffs und damit eine schnellere Abarbeitung der XPath-Anfrage ergeben.

- Full Text Indexing: XPath-Anfragen, in denen Funktionen wie „contains“ enthalten sind, d.h. die nach einem String in einem Textknoten suchen, könnten dadurch beschleunigt werden, dass full text indexing (FTI) in der Datenbank unterstützt wird. Das FTI bezeichnet einen Transformationsprozess, bei dem Text derart umstrukturiert wird, dass er für einen späteren Suchvorgang optimiert vorliegt [The04].
  Hier wäre es vorstellbar, alle Textknoten zu indizieren, um ein Durchsuchen der Textknoten zu beschleunigen. Dabei sollte das Indizieren möglichst unabhängig von der darunter liegenden Datenbank erfolgen, da die Middleware auf möglichst allen relationalen Datenbanken arbeiten können soll. Eine Möglichkeit dazu bietet das Open-Source-Framework Lucene [Apa06a].
  Es soll geprüft werden, ob eine Volltext-Indizierung der Textknoten mit Lucene möglich ist und untersucht werden, wie FTI in die XPath Engine Jaxen integriert werden kann.

4.3 Zuverlässigkeit

4.3.1 Testen

Die Zuverlässigkeit der neuen Version der Middleware soll durch eine komplette Abdeckung einzelner Klassen mit JUnit-Tests gesichert werden.

4.3.2 Analyse des Codes

In der neuen Version der Middleware soll der Quellcode in einer bereinigten Form vorliegen. Das heißt, dass der Code nach Coding-Standards und auf Bugs

untersucht werden soll und erforderliche Änderungen durchgeführt werden sollen.
Dazu sollen folgende Tools zum Einsatz kommen:

- Checkstyle[Bur06]: Der Quellcode wurde mit dem Werkzeug Eclipse entwickelt. Deshalb bietet es sich an, auch die Analyse des Codes mit diesem Werkzeug zu machen. Eclipse bietet diese Funktion von Hause aus nicht an, kann aber durch das Open-Source Plug-in Checkstyle erweitert werden.
  Dieses Plug-in sichert die Einhaltung einer Menge von Coding-Standards, indem es den Quellcode zur Kompilierzeit analysiert und Stellen, an denen diese Standards verletzt wurden, markiert.

- IntelliJ[Jet06]: Die Entwicklungsumgebung IntelliJ beinhaltet bereits eine Funktion, die „Inspect Code“ heißt und den Quellcode beim Kompilieren wie auch Checkstyle auf verschiedene Aspekte untersucht und je nach Einstellung mit einer Fehlermeldung oder Warnung darauf hinweist.
  Da bei dieser Analyse teilweise andere Aspekte als bei Checkstyle betrachtet werden, soll der Code auch hiermit analysiert werden. Eine Testversion von IntelliJ steht für 30 Tage zur Verfügung.

- FindBug: FindBug [Sou06b] sucht nach Bugs in Java-Code und basiert dabei auf dem Konzept von Bug Patterns. Ein Bug Pattern ist ein Ausdruck im Code, der oft zu einem Fehler führt. Die Analyse läuft auch hier statisch ab. Das Programm ist als freie Software verfügbar und kann ebenfalls als Plug-in in Eclipse integriert werden.

5 Umsetzung

Im folgenden Kapitel wird erläutert, wie die Anforderungen an die neue Version der Middleware tatsächlich umgesetzt wurden, welche Probleme auftauchten und wie diese gelöst wurden.

5.1 Funktionsumfang

5.1.1 Importieren und Exportieren von XML-Dokumenten

Die in Kapitel 4.1.2 beschriebene Modifizierung des EdgeInline-Schemas hat zunächst Auswirkungen auf das Einlesen von XML-Dokumenten in die Datenbank. Bisher wurden XML-Dateien unter Verwendung eines SAX[5][Sou06a]-Parsers eingelesen und nach dem „alten“ EdgeInline-Schema in der Datenbank gespeichert.
SAX benutzt das Observer Pattern [Gam95], um Client-Applikationen mitzuteilen, wie der Inhalt eines XML-Dokuments aussieht. Dabei übernimmt der Parser die Rolle des Subjekts, welches Ereignisse empfängt und an den Client weiter gibt. Dieser übernimmt die Rolle des Observers, der entsprechend auf die Ereignisse reagiert [Har03].
Ein solches Ereignis stellt beispielsweise die Methode `startElement(String uri, String localName, String qName, Attributes atts)` dar, welche das Start-Tag eines Elements mit der Namensraum-URI „uri“, dem lokalen Namen „localName“, dem qualifizierten Namen „qName“ und den Attributen „atts“ meldet.
Der Client muss das Interface `org.xml.sax.ContentHandler` implementieren, um auf die Ereignisse zu reagieren und muss vorher beim SAX-Parser registriert worden sein.
Importiert man ein XML-Dokument, wird zuerst eine neue, leere Resource erzeugt. Dann wird der Inhalt der Resource gesetzt.
Dazu wird in der Middleware die Methode `setContent` in der Klasse `XMLResourceImpl` aufgerufen. In der Implementierung dieser Methode wird ein SAX-Parser instanziiert. Ein Objekt der Klasse `EdgeInline` wird beim Parser registriert. Der Parser liest die XML-Datei ein und meldet die Ereignisse an `EdgeInline`. Diese Klasse ist eine Implementierung des ContentHandlers, die auf Ereignisse reagiert, indem sie den jeweiligen Knoten in der Datenbank speichert.
Die Erweiterung dieser Klasse auf andere Knotentypen bringt das Problem mit sich, dass der SAX-Parser keine Ereignisse für CDATA-Abschnitte, Entity-Referenzen und Kommentare bereitstellt. Diese Konstrukte sollen in der neuen Version der Middleware aber auch in der Datenbank gespeichert werden.
Eine erster Lösungsansatz war, statt SAX den DOM-Parser zu verwenden, da DOM bereits als Orientierung für das neue Mapping-Schema diente und somit alle von uns benötigten Konstrukte erkennt.
Das DOM hat allerdings den schwerwiegenden Nachteil, dass es das gesamte Dokument als DOM-Baum darstellt, was bei großen Dokumenten zu Speicherengpässen führt [Nie04].

[5] Simple API for XML

Der SAX-Parser bietet den Vorteil, dass er das Dokument als sequentiellen Datenstrom einliest und so effizienter arbeitet und wenig Speicher benötigt [Sou06a].
Um die neuen Konstrukte ebenfalls als Ereignisse vom SAX-Parser melden zu lassen, muss eine Erweiterung von SAX genutzt werden, die es erlaubt, auch lexikalische Informationen eines XML-Dokuments bereitzustellen [Sou06c]. Dazu muss zusätzlich zum `ContentHandler` das Interface `LexicalHandler` implementiert werden und mit `setProperty` beim Parser registriert werden. Der Parser erzeugt dann auch Ereignisse für Konstrukte wie Kommentare, Entity-Referenzen und CDATA-Abschnitte.
Der neuen Version der Middleware wurde die Klasse `LexicalHandlerImpl` hinzugefügt, die den `LexicalHandler` implementiert, so dass nun auch diese Konstrukte in der Datenbank gespeichert werden.
In der Klasse `EdgeInline` wurde das Anlegen und Befüllen der Datenbanktabelle angepasst. Die Spalte *attribute* wurde durch die Spalte *type* ersetzt, deren Typ auf `varchar(50)` festgelegt wurde.
Die Spalte *src* wurde komplett entfernt.
Das Speichern von Elementen wurde so verändert, dass einem Element nun kein Wert (Spalte *val*) mehr zugeordnet wird. Textknoten werden als separate Tupel in der Datenbank gespeichert.
Beim Einlesen werden nun auch Namensräume berücksichtigt, worauf in Kapitel 5.1.3 noch genauer eingegangen wird.
Des Weiteren wurde das Exportieren einer Resource in eine XML-Datei in der Klasse `T1_EdgeInline` an das neue Schema angepasst.

5.1.2 Anpassung des Jamrox-DOM-Datenmodells

Die Änderung des EdgeInline-Schemas hat Auswirkungen auf die DOM-Darstellung der Datenbank, die für Jaxen simuliert wird. Dies hatte zur Folge, dass alle Klassen im Paket `org.jaxen.jamrox` verändert werden mussten.
Das Paket enthält zusätzlich zu den Jamrox-DOM-Klassen die Klasse `JamroxSqlConnection`, welche die Verbindung zur Datenbank liefert und die prepared Statements enthält. Letztere mussten angepasst werden. Hier befand sich in einigen Statements die Bedingung `attribute="false"`. Da nun keine Spalte *attribute* mehr existiert, sondern die Spalte *type*, die den Knotentyp enthält, wurde diese Bedingung durch `type!="attribute"` ersetzt.
Zusätzlich wurde hier folgender Fehler beseitigt: Den Statements, die den „previous“ bzw. „next sibling“ eines Knotens liefern, fehlte die Bedingung, dass keine Attribute im Ergebnis enthalten sein dürfen. Das Fehlen dieser Bedingung führt dazu, dass auch Attribute bei einer entsprechenden XPath-Anfrage ausgegeben werden, wenn die ord-Zahl zufällig der eines „siblings“ entspricht. Der eigentliche „sibling“ wird dann nicht berücksichtigt. Dies wurde durch Einfügen der Bedingung `type!="attribute"` behoben.
Die Klassen `JamroxText` und `JamroxElement` wurden dahingehend verändert, dass Text-Knoten nun eigenständige Knoten darstellen und nicht mehr als zum Parent-Element-Knoten gehörender Wert gespeichert werden.
Außerdem konnten nun auch die Klassen

`JamroxComment`,
`JamroxProcessingInstruction`,
`JamroxCDATA` und
`JamroxEntityReference`

implementiert werden. Dies war in der Version nach [Sie05] nicht möglich, da diese Konstrukte noch nicht in der Datenbank gespeichert wurden.
Diese „neuen“ Knoten, die nun von der Middleware unterstützt werden, hatten wiederum Auswirkungen auf fast alle schon vorhandenen Klassen des Pakets `org.jaxen.jamrox`, so dass auch hier Anpassungen notwendig waren. Wenn nun z.B. in der Klasse `JamroxElement` bei `getNextSibling()` auf ein Datenbank-Tupel zugegriffen wird, wird nun immer unterschieden, ob es sich um ein Element,- ein Comment,- ein Text,- ein CDATA, ein Processing-Instruction- oder ein Entity-Referenz-Tupel handelt, um dann den entsprechenden DOM-Knoten zu erzeugen.

5.1.3 Namensraum-Unterstützung

In der Version nach [Sie05] wurden Elemente und Attribute nur mit ihrem lokalen Namen in der Datenbank abgelegt. In der neuen Version wird in der Spalte *name* nun der qualifizierte Name gespeichert.
Außerdem werden Namensraumdeklarationen nun ebenfalls als Attribute in der Datenbank abgelegt.
Wie bereits in Kapitel 4.1.2 erläutert, sollen die Namensraumdeklarationen zusätzlich in einem Java-Objekt gespeichert werden, um einen schnelleren Zugriff auf die URI eines Elements oder Attributs zu ermöglichen. Dies wurde mit Hilfe einer Java-HashMap umgesetzt, die als statisches Attribut der Klasse `JamroxNode` hinzugefügt wurde. Eine HashMap ist ein Objekt, das Abbildungen von Schlüsseln auf Werte speichert, d.h. in diesem Fall Abbildungen von Präfixen auf URIs.
Bevor auf dem Jamrox-DOM-Baum navigiert werden kann, muss immer erst ein `JamroxDocument` erzeugt werden. Aus diesem Grund wird das Befüllen der HashMap mit Namensraumabbildungen im Konstruktor von `JamroxDocument` angestoßen. Anschließend kann mit der zur HashMap gehörenden `get`-Methode auf die Namensraumabbildungen zugegriffen werden.
Dieser Zugriff ist immer dann nötig, wenn die URI eines Elements oder Attributs benötigt wird. Damit der Zugriff auf die HashMap pro `JamroxNode` nur einmal erfolgen muss, wurde die URI als Attribut `uri` in `JamroxNode` hinzugefügt. In den Klassen `JamroxElement` und `JamroxAttr` wird in den Konstruktoren die zum Präfix gehörende URI diesem Attribut zugeordnet, bei allen anderen JamroxKlassen bleibt die URI leer. So kann in der Methode `getNamespaceURI()` direkt auf das `uri`-Attribut zugegriffen werden.
Die Methoden `getLocalName()` und `getNodeName()` konnten nun auch gemäß der DOM-Spezifikation implementiert werden, so dass sie den lokalen Namen bzw. den qualifizierten Namen eines Elements oder Attributs liefern.
Bei der Implementierung dieser Methoden wurde darauf geachtet, dass bei Elementen, die in einem Default-Namensraum liegen, das „konstruierte“ Präfix (siehe Kapitel 4.1.2) entfernt wird.
Möchte man nun mit Jaxen XPath-Anfragen stellen, in denen Elemente oder

Attribute enthalten sind, die einem Namensraum zugeordnet sind, muss folgendes beachtet werden: Jaxen weiß nicht, welche Namensräume in dem betreffenden XML-Dokument deklariert wurden. Deswegen müssen die in der XPath-Anfrage enthaltenen Präfixe an die zugehörigen URIs gebunden werden und Jaxen bekannt gemacht werden. So wird ein „Namensraum-Kontext“ geschaffen, mit dem Jaxen dann arbeitet [Cod06a].
Dies geschieht mit der Methode `addNamespace(String prefix, String uri)`, mit der ein Präfix-URI-Paar deklariert wird.
Um diese Funktionalität dem Nutzer zugänglich zu machen, enthält die XML:DB-API im Interface `XPathQueryService` folgende Methoden [XML06f]:

- `setNamespace(String URI, String prefix)`: fügt eine Namensraum-Abbildung zur internen Namensraum-Tabelle hinzu
- `getNamespace(String prefix)`: gibt die URI von der internen Namensraum-Tabelle zurück, die mit „prefix“ assoziiert wurde
- `removeNamespace(String prefix)`: entfernt die Namensraum-Abbildung, die mit „prefix“ assoziiert wurde
- `clearNamespaces()`: löscht alle Namensraum-Abbildungen

Zur Implementierung dieser Methoden, wurde in der neuen Version der Middleware in der Klasse `XPathQueryService` eine HashMap angelegt, welche als „interne Namensraumtabelle“ fungieren soll.
Die oben genannten Methoden wurden so implementiert, dass sie die HashMap gemäß Spezifikation modifizieren bzw. abfragen.
Bevor eine XPath-Anfrage gestellt wird, können mit diesen Methoden die in der XPath-Anfrage vorkommenden Präfixe an die zugehörigen URIs gebunden werden.
Wird dann die XPath-Anfrage gestellt, wird in der Implementierung der Methode `queryResource` zuerst jede in der HashMap vorkommende Abbildung mit der eben erläuterten Jaxen-Methode `addNamespace` dem Kontext der XPath-Anfrage hinzugefügt.
Falls die XPath-Anfrage Elemente enthält, die einem voreingestellten Namensraum zugehörig sind, muss dieser Default-Namensraum ebenfalls zum Kontext hinzugefügt werden.
Dies geschieht genauso wie oben beschrieben, nur dass hier ein eindeutiges Präfix „erfunden“ werden muss, welches der URI zugeordnet wird. Dieses Präfix muss in der XPath-Anfrage den Elementen, die dem voreingestellten Namensraum zugeordnet sind, vorangestellt werden.
Durch die Namensraum-Unterstützung kann nun auch die Namespace-Achse im XPathQueryService genutzt werden. Namespace-Deklarationen werden zwar in DOM und deswegen auch in Jamrox wie Attribute behandelt, können allerdings mit XPath gesondert mit der Namespace-Achse abgefragt werden.
Deswegen gibt es im Paket `jaxen.org.jamrox` zusätzlich zu den „normalen“ DOM-Klassen noch die Klasse `JamroxNamespaceNode`. Sie stellt die Namespace-Deklarationen als „spezielle“ Attribute dar und wird beim Aufruf der Namespace-Achse genutzt.

5.1.4 DOM- und SAX-Unterstützung

Damit der Inhalt einer Resource auch als DOM-Baum oder in Form von SAX-Events zugänglich gemacht werden kann bzw. der Inhalt einer Resource mit SAX oder DOM gesetzt werden kann, wurden in der neuen Version der Middleware die folgende Methoden implementiert [XML06e]:

- `Node getContentAsDOM()`: Der Inhalt einer Resource soll als DOM-Knoten zurückgegeben werden.
  In der Implementierung musste nur zu der Resource ein `JamroxDocument` erzeugt und zurückgegeben werden. Die Darstellung der Datenbank als DOM-Modell ist für Jaxen bereits vorhanden. Über das `JamroxDocument` kann dann wie auf einen DOM-Baum zugegriffen werden.

- `void setContentAsDOM(Node node)`: Der Inhalt einer Resource soll auf den Inhalt von „node“ gesetzt werden.
  Für diese Funktionalität wurde eine eigene Klasse `DOMHandler` eingeführt. In dieser Klasse wird ein `Iterator` definiert, der über den DOM-Baum, der als Eingabeparameter mitgegeben wurde, iteriert. Dabei wird jeder Knoten dieses DOM-Baums in der Resource und damit in der Datenbank gespeichert.

- `void getContentAsSAX(ContentHandler handler)`: Der Inhalt der Resource wird geparst und in Form von SAX-Events an „handler“ gemeldet.
  Für diese Methode musste ein Parser geschrieben werden, der in die Klasse `SAX_DB_Parser` ausgelagert wurde. In dieser Klasse werden die Hierarchieebenen der Resource rekursiv durchlaufen, deren Inhalt als SAX-Events geliefert werden soll. Die Hierarchieebenen sind über die gleiche parent-Id erkennbar. Für jedes Tupel wird das zugehörige SAX-Event ausgelöst, d.h. die entsprechende Methode des ContentHandlers „handler“ aufgerufen.
  Wenn der „Parser“ z.B. auf ein Tupel vom Typ „Element“ stößt, wird das Event `startElement(String uri, String localName, String qName, Attributes atts)` erzeugt. Dabei ist zu beachten, dass keine Ereignisse für lexikalische Bestandteile (wie Kommentare usw.) ausgelöst werden, da diese nur an einen `LexicalHandler` gemeldet werden könnten. Hier ist aber nur ein „normaler“ ContentHandler vorhanden.

- `ContentHandler setContentAsSAX()`: Es soll ein ContentHandler zurückgegeben werden, der den Inhalt einer Resource gemäß den vorkommenden Ereignissen setzt.
  Das heißt, dass in der Implementierung des ContentHandlers z.B. in der Methode `startElement(String uri, String localName, String qName, Attributes atts)` der Resource ein Element mit den dazugehörenden Attributen hinzugefügt wird.
  Diese Funktionalität steht bereits im ContentHandler `EdgeInline` zur Verfügung, der für die Methode `setContent` implementiert wurde.
  Das bedeutet, dass wir in dieser Methode ein Objekt der Klasse `EdgeInline` zurückliefern können. Auch hier muss beachtet werden, dass mit diesem ContentHandler nur die „normalen“ SAX-Ereignisse behandelt werden.

Für Kommentare, CDATA-Abschnitte usw. müsste zusätzlich ein LexicalHandler zurückgeliefert werden. Um die Funktionalität der Klasse `EdgeInline` besser zu beschreiben, wurde sie in `SAXHandler` umbenannt.

5.1.5 Hinzufügen des TransactionService und des XUpdateQueryService

Für die Implementierung des TransactionServices war bei den XML:DB-API-Methoden `commit()` und `rollback()` nur ein Aufrufen der gleich lautenden JDBC-Methoden notwendig. In der Implementierung der XML:DB-API-Methode `begin()` wird der AutoCommit-Modus ausgeschaltet und in `commit()` werden die Änderungen festgeschrieben und am Schluss der AutoCommit-Modus wieder eingeschaltet.
Für den XUpdateQueryService wird von der XML:DB Working Group [XML06a] die Sprache XUpdate bereitgestellt, die Syntax und Semantik für verschiedene Update-Konstrukte spezifiziert.
Nach intensiver Auseinandersetzung mit der Update-Sprache wurde deutlich, dass die Spezifikation von XUpdate einige Unstimmigkeiten und Lücken aufweist. Später wird auf diese noch eingegangen.
XUpdate ist kein W3C-Standard. Eine XML-Update-Sprache des W3C befindet sich noch im Standardisierungsprozess [W3C06f]. Diese Update-Sprache heißt XQueryUpdate und soll eine Erweiterung von XQuery sein, so dass es möglich ist, Änderungen an einer Instanz des XQuery/XPath 2.0-Datenmodells durchzuführen [W3C06f].
Da die API bisher nur auf XUpdate ausgerichtet ist, wurde der XUpdateQueryService mit dieser Sprache implementiert. Falls der XUpdateQueryService der API nach Fertigstellen des W3C-Standards auf die Sprache XQueryUpdate ausgerichtet wird, sollte der XUpdateQueryService in weiterführenden Arbeiten dahingehend überarbeitet werden.
Im Folgenden soll die Sprache XUpate kurz erläutert und auf die gefundenen Schwächen hingewiesen werden. Ein Update bzw. mehrere Updates werden in einem XML-Dokument festgehalten. In diesem Dokument befinden sich ein oder mehrere XUpdate-Konstrukte, die beschreiben, wie das XML-Dokument verändert werden soll. Um XML-Knoten auszuwählen, wird XPath benutzt.
An einem Beispiel soll die Funktionsweise von XUpdate erklärt werden. Die XML-Datei in Abbildung 8 soll verändert werden. Das Update-Dokument zeigt Abbildung 9.
Jedes XUpdate-Dokument beginnt mit dem `<xupdate:modifications/>`-Element. Das darauf folgende Element `<xupdate:insert-after/>` beschreibt eine Art eines Updates, nämlich das Einfügen eines neuen Inhalts als Geschwisterknoten **nach** einem Referenzknoten. Der Referenzknoten wird mit dem `select`-Attribut ausgewählt, das als Wert einen XPath-Ausdruck enthält. Es gibt auch Konstrukte, die einen neuen Inhalt als vorhergehenden Geschwisterknoten (`xupdate:insert-before`) oder als Kindknoten (`xupdate:append`) einfügen [XML06g].
Mit dem Attribut `select` wird im `<xupdate:insert-after/>`-Element das erste `address`-Element der „adresses“-Datei ausgewählt.
Der erste Kindknoten von `insert-after` ist ein `<xupdate:element/>`-Element.

```
<?xml version="1.0"?>
<addresses version="1.0">
   <address id="1">
      <fullname>Andreas Laux</fullname>
      <born day="1" month="12" year="1978"/>
      <town>Leipzig</town>
      <country>Germany</country>
   </address>
</addresses>
```

Abbildung 8: XML-Datei „addresses“ [XML06g]

```
<?xml version="1.0"?>
<xupdate:modifications version="1.0"
   xmlns:xupdate="http://www.xmldb.org/xupdate">

   <xupdate:insert-after select="/addresses/address[1]" >
      <xupdate:element name="address">
         <xupdate:attribute name="id">2</xupdate:attribute>
         <fullname>Lars Martin</fullname>
         <born day="2" month="12" year="1974"/>
         <town>Leizig</town>
         <country>Germany</country>
      </xupdate:element>
   </xupdate:insert-after>

</xupdate:modifications>
```

Abbildung 9: XUpdate-Datei [XML06g]

Dieser Ausdruck erzeugt ein Element, das den Namen des Wertes des Attributs `name` erhält, in diesem Fall also „address“.
Hier wird bereits eine Schwäche deutlich, da beim Erzeugen eines Elements keine Angaben über den Namensraum gemacht werden können.
Das Erzeugen von Attributen funktioniert genauso wie das von Elementen, nur das hier zusätzlich der Wert des Attributs zwischen Start- und Ende-Tag angegeben werden muss. Auch das Erzeugen von Texten, Kommentaren und Processing-Instructions läuft analog ab.
Nach der Erzeugung des Attributs mit Namen „id“ und Wert „2“ wird der einzufügende Inhalt als normaler XML-Code angegeben. Das heißt, der Code soll genauso in die veränderte „addresses“-Datei übernommen werden.
Es ist widersprüchlich, dass man den einzufügenden Inhalt auch auf diese Weise angeben kann, da die vorher gezeigten `xupdate`-Konstrukte zum Erzeugen von Knoten dann eigentlich überflüssig sind.
Problematisch wurde dies auch beim Parsen der XUpdate-Datei, worauf aber später bei der Erklärung der Implementierung des XUpdateQueryServices noch genauer eingegangen wird.

```
<?xml version="1.0"?>
<addresses version="1.0">
   <address id="1">
      <fullname>Andreas Laux</fullname>
         <born day="1" month="12" year="1978"/>
         <town>Leipzig</town>
         <country>Germany</country>
   </address>
   <address id="2">
      <fullname>Lars Martin</fullname>
      <born day="2" month="12" year="1974"/>
      <town>Leizig</town>
      <country>Germany</country>
   </address>
</addresses>
```

Abbildung 10: XML-Datei „addresses“ [XML06g] nach dem Update

Die Datei sieht nach dem Update dann so aus wie in Abbildung 10 dargestellt. Neben den gerade erklärten XUpdate-Konstrukten gibt es noch folgende:

- `xupdate:update` soll den Inhalt eines Knotens „updaten“. Als Inhalt ist hier der Textknoten gemeint, der sich zwischen Start- und Ende-Tag des Elements befindet.
  Diese Definition von Inhalt oder Wert eines Elementknotens findet sich weder in der DOM- noch im XPath-Datenmodell wieder. In DOM beispielsweise ist der Wert eines Elements immer `null`. Sinnvoller wäre es, den Textknoten auszuwählen und diesen „upzudaten“.

- `xupdate:remove` erlaubt einen Knoten zu entfernen. Hierbei wird in der Spezifikation nicht festgelegt, ob die Kindknoten dabei ebenfalls entfernt werden.

- `xupdate:rename` verändert den Namen eines Elements oder Attributs. Auch hier wird nichts über Namensräume gesagt. [XML06g]

Außerdem gibt es ein Konstrukt zur Deklaration von Variablen (`xupdate:variable`), wobei nicht klar festgelegt ist, an welchen Stellen der Wert der Variable eingesetzt werden kann.
Die Methode `updateResource(String id, String commands)` des XUpdateQueryServices wurde folgendermaßen implementiert: Das Update-Dokument wird geparst und für jeden gefundenen XUpdate-Ausdruck wird eine entsprechende DOM-Methode aufgerufen, die das `JamroxDocument` der zu verändernden Resource „updatet“.
In den DOM-Interfaces gibt es einige Methoden, die eine Manipulation der Knoten des DOM-Dokuments beschreiben (z.B. `appendChild(Node newNode)`). Für Update-Konstrukte, für die es keine entsprechenden DOM-Methoden gibt, wurden eigens definierte Methoden hinzugefügt.
Die DOM-Update-Methoden wurden so implementiert, dass das Update in der

Datenbank vollzogen wird. Hierbei galt es zu beachten, dass die Reihenfolge der Knoten konsistent bleibt und dass beim Löschen eines Elements auch alle Referenzen auf dieses Element gelöscht werden.
Das Parsen der Update-Datei wurde mit dem StAX[6]-Parser [Bea06] umgesetzt. Im Gegensatz zu den „Push"-APIs wie SAX stellt StAX eine „Pull"-API dar, bei der das **Client-Programm** den Parser nach dem nächsten Stück Inhalt fragt, nicht umgekehrt. Diese APIs stellen eine komfortable Alternative zu den Push-APIs dar. StAX ist wie SAX fähig, große Dokumente zu lesen, ermöglicht es aber anders als SAX, dass die **Applikation** die Steuerung übernimmt [Bea06].
Da es in XUpdate starke Abhängigkeiten zwischen Elementen und ihren Nachfolgerknoten gibt, ist das Parsen der Update-Datei mit StAX einfacher zu lösen als mit SAX, da hiermit ganz gezielt z.B. nach dem Nachfolger-Element gefragt werden kann und in Abhängigkeit davon bestimmte Reaktionen ausgelöst werden können. Würde das Parsen mit SAX durchgeführt, müssten vorangegangene Ereignisse (wie z.B. das Auftreten eines bestimmten Elements) immer zwischengespeichert werden.
Im Folgenden soll erläutert werden, welche Unklarheiten und Lücken in der Spezifikation von XUpdate zu Problemen bei der Umsetzung des XUpdate-QueryServices führten und wie diese gelöst wurden:

- Oben wurde bereits angedeutet, dass das Vorkommen von beliebigem XML-Code in der Update-Datei beim Parsen Probleme bereitete. Dies liegt daran, dass das Parsen dieses XML-Codes ähnlich wie in der EdgeInline-Klasse geschehen müsste, da es keine feste Syntax wie bei den übrigen Update-Konstrukten gibt.
  Außerdem ist das Benutzen von XML-Code unnötig, da es für das Erzeugen von allen Knoten entsprechende Ausdrücke gibt. Aus diesem Grund wurde der Fall des Vorkommens von XML-Code in der Update-Datei bei der Implementierung nicht berücksichtigt. Die Update-Datei soll nur aus XUpdate-Ausdrücken bestehen.

- Die fehlende Unterstützung von Namensräumen im XUpdateQueryService wurde ausgeglichen, indem beim Hinzufügen eines Elements untersucht wird, ob der Name des neu erzeugten Elements mit einem Präfix beginnt. Trifft dieses zu, wird zum Präfix die zugehörige URI in den Namensraumabbildungen (siehe Kapitel 5.1.3) gesucht und dem Element zugeordnet. Ist das Präfix nicht deklariert worden, wird eine Fehlermeldung ausgegeben.
  Das Erzeugen von Elementen, die in einem Default-Namensraum liegen, ist allerdings nicht möglich, da es kein Attribut im `xupdate:element`-Ausdruck gibt, in dem die URI angegeben werden könnte. Ob das Element in einem Namensraum liegt, kann nur durch das Vorhandensein eines Präfixes im Namen erkannt werden.

- Das `xupdate:update`-Konstrukt widerspricht zwar dem DOM-Datenmodell, wurde aber so umgesetzt, dass das erste Kind eines Elements, welches ein Textknoten ist, verändert wird.

[6]Streaming API for XML

- Mit `xupdate:remove` werden alle Kinder, Kindeskinder usw. eines Elements mitgelöscht.
- Da die Verwendung von Variablen nicht klar spezifiziert wurde, wurde die Implementierung dieses Konstrukts weggelassen.

Als Rückgabewert liefert die Methode `updateResource` die Anzahl der Veränderungen in einem Dokument.
Die zweite Methode des XUpdateQueryServices `update(String commands)` ruft für alle Resourcen in der ausgewählten Collection die Methode `updateResource` auf.

5.1.6 Validierung

Ergänzend zum XUpdateQueryService wurde die Durchführung einer Validierung gefordert (siehe Kapitel 3.1.8). Dies bedeutet, dass bevor das Update durchgeführt wird, überprüft wird, ob das Dokument nach der Änderung noch den Anforderungen der dazugehörigen DTD oder dem XML-Schema (falls vorhanden) entspricht.
Zur Realisierung dieser Funktionalität bietet sich, wie bereits in Kapitel 4.1.3 dargestellt, die Benutzung der Bibliothek `org.w3c.dom.validation` an. Die Schnittstellen werden bisher nicht als lpgl[7] [Nov06]-konforme Implementierung angeboten [W3C06b]. Die lpgl-Lizenz garantiert, dass die Software für alle Benutzer frei ist [Nov06].
Die Implementierung der Firma Oracle [Ora06] stand für Tests allerdings zur Verfügung.
Unter der Voraussetzung der Kapselung der Oracle-Bibliotheken, so dass es ohne Umstände wieder möglich wäre, diese durch andere zu ersetzen, hätten diese eingesetzt werden können. Dem Einsatz der Bibliotheken stellten sich allerdings folgende Probleme entgegen:
Die Validierungs-Methoden (`canInsertBefore` usw.) können nur auf der DOM-Implementierung von Oracle angewendet werden. Diese werden mit `XMLDocument`, `XMLElement` usw. bezeichnet. Das Aufrufen der Validierungs-Methoden mit den Jamrox-DOM-Klassen ist möglich, wenn die Oracle-DOM-Klassen als Oberklassen definiert werden. Danach können die Validierungsmethoden zwar aufgerufen werden, liefern aber keine richtigen Ergebnisse.
Dies liegt daran, dass in der DOM-Implementierung von Oracle eigene Attribute, Flags und Methoden definiert und implementiert werden, die über die DOM-Spezifikation hinausgehen. Das Funktionieren der Validierungs-Methoden ist von diesen Modulen abhängig.
Die spezielle Ausrichtung der Implementierung von Oracle wird gut erkennbar, wenn man die Erzeugung eines Oracle-spezifischen DOM-Documents betrachtet. Dazu bietet Oracle einen **speziellen** Parser, der aus einer XML-Datei ein `XMLDocument` erstellt. Diesem Document kann ein XML-Schema zugeordnet werden [Voo06]. Diese Funktionalität ist nicht in der DOM-Spezifikation enthalten.
Dies führt zur Schlussfolgerung, dass es nicht möglich ist, die Oracle-Bibliotheken

[7] lesser general public license

mit dem in Jamrox implementierten DOM-Modell zu verwenden.
Da es keine andere zur Verfügung stehende Implementierung der DOM-Validation-Spezifikation gibt und es den Umfang dieser Arbeit übersteigen würde, die Schnittstellen selbst zu implementieren, konnte die inkrementelle Überprüfung der Gültigkeit in dieser Arbeit nicht umgesetzt werden.
In weiterführenden Arbeiten könnte die Implementierung entweder selbst vorgenommen werden oder, falls dann eine lpgl-konforme Implementierung zur Verfügung steht, diese in den XUpdateQueryService integriert werden.

5.1.7 XML:DB-GUI

Wie bereits in Kapitel 3.1.1 gefordert und in Kapitel 4.1.1 beschrieben, soll die Middleware an eine Anwendung angebunden werden, die eine grafische Oberfläche für XML:DB-API konforme Implementierungen bereitstellt. Hierbei gab es in vorangegangenen Arbeiten Probleme, die XML:DB-GUI so zu konfigurieren, dass die Anbindung an die Middleware funktioniert.
Die Konfiguration der XML:DB-GUI wird in einer XML-Datei festgehalten, die für jede XML-Datenbank, die an die GUI angebunden werden soll, einen Eintrag enthält. Für die Jamrox-Middleware sieht die richtige Konfiguration folgendermaßen aus:

```
<XMLDBConfig name="jamrox">
   <Driver>jamrox.xmldbapi.DatabaseImpl</Driver>
   <Uri>xmldb:jdbc:postgresql://localhost/bsp</Uri>
   <RootCollection>/rootName</RootCollection>
   <!--hier folgen noch Einträge für die Services-->
</XMLDBConfig>
```

Das `Driver`-Element verweist auf die Klasse `DatabaseImpl`. Dies ist die Implementierung der API-Klasse `Database`, die in der Spezifikation folgendermaßen beschrieben wird:

> `Database` ist die Abkapselung der Datenbank-Treiber-Funktionalität, die notwendig ist, um auf eine XML-Datenbank zuzugreifen. Die Implementierung wird beim `DatabaseManager` registriert, um Zugriff auf die Resourcen in der XML-Datenbank zu ermöglichen [XML06b].

Eine wichtige Methode in dieser Klasse ist `getCollection`, da der `DatabaseManager` diese aufruft, um einen erstmaligen Zugriff auf die root-Collection zu erlangen.
Der Pfad zur Datenbank wird im `Uri`-Element in der Konfiguration angegeben, wobei hier das Präfix „xmldb:“ vorangestellt werden muss.
Danach muss der Pfad zur root-Collection angegeben werden. Dieser ist relativ zum Pfad, der im `Uri`-Element steht.
Zum Schluss werden noch die zur Verfügung stehenden Services registriert (z.B. XPathQueryService). Optional können Benutzername und Passwort angegeben werden, falls dies für die Anmeldung bei der Datenbank nötig ist [Tit06].
Der Grund, warum die Anbindung an den GUI-Client nicht funktionierte, lag

allerdings nicht in der Konfiguration, sondern in der Jamrox-Implementierung der XML:DB-API. Hier führten einige Fehler in der Implementierung von `DatabaseImpl` dazu, dass der Zugriff auf die Datenbank für die GUI nicht möglich war.
Wie oben in der Spezifikation deutlich wird, ist dies die Klasse, mit der der Zugriff auf die Datenbank erst ermöglicht wird.
Der erste Fehler in `DatabaseImpl` war folgender: In dieser Klasse gibt es ein statisches Attribut `INSTANCE_NAME`, das den Namen der Datenbank enthalten soll, d.h. in unserem Fall „jdbc“.
Dieses Attribut war in der Version nach [Sie05] auf „xmldb:/“ gesetzt. Dies führte zum Scheitern des Verbindungsaufbaus, denn das „xmldb“-Präfix wird vom `DatabaseManager` direkt wieder entfernt und so das zweite Token aus der Datenbank-Uri mit diesem Attribut verglichen. Sind die beiden Werte nicht gleich, wird die Verbindung nicht aufgebaut.
Ein weiterer Fehler in der Klasse `DatabaseImpl` befand sich in der Methode `getCollection(String uri, String username, String password)`, die vom `DatabaseManager` aufgerufen wird, um die root-Collection zu erhalten.
Der Eingabeparameter `uri` enthält den Pfad zur Datenbank konkateniert mit dem Pfad zur Root-Collection (beides gelesen aus der Konfigurations-Datei). Um die geforderte Collection zurückzugeben, muss zuerst der Datenbankpfad aus `uri` entfernt werden.
Hier wurden in der Version nach [Sie05] nur die ersten 7 Zeichen entfernt (wahrscheinlich für „xmldb:/“). Dies wurde geändert, so dass nun die ersten 4 Tokens, die durch „/“ getrennt sind, (z.B. „jdbc/postgresql/localhost/bsp“) entfernt werden. Anschließend wird mit dem Pfad zur Collection (z.B.“/rootname“) der Konstruktor von `CollectionImpl` aufgerufen.
Mit diesen Änderungen konnte die GUI mit der Jamrox-Konfiguration gestartet werden. Es traten dann allerdings noch weitere Fehler beim Benutzen der GUI auf, die häufig mit einer nicht der Spezifikation entsprechenden Implementierung der XML:DB-API zu tun hatten.
In der folgenden Aufzählung sollen diese Fehler beschrieben werden und wie sie behoben wurden.

- In der Klasse `CollectionImpl` wird in der Methode `createResource` als Eingabeparameter der Typ der Resource mitgegeben, da es möglich ist, verschiedene Arten von Resourcen zu definieren. Der Typ wird von der GUI auf den Wert „XMLResource“ gesetzt, wenn eine Resource erzeugt werden soll. So ist es auch in der API vorgegeben.
  In der Implementierung der Methode fand in der Version nach [Sie05] eine Abfrage nach dem Typ der Resource und der Vergleich mit dem String „XML_RESOURCE“ statt. Da die Strings nicht übereinstimmten, wurde die Resource nicht erzeugt. Der Typ wurde in der neuen Version auf „XMLResource“ berichtigt.

- Beim Erzeugen einer Resource wurde diese fälschlicherweise zweimal in die `MetaDocumentTable` eingetragen. Das erste Mal im Konstruktor von `XMLResource`, das zweite Mal in der Methode `createResource` in der Klasse `CollectionImpl`. Dies führte zur Fehlermeldung, die Resource sei bereits vorhanden. In der neuen Version wurde das zweite Hinzufügen

entfernt.

- Möchte man in der GUI eine XMLResource anzeigen, kann dies mit Doppelklick auf die Resource geschehen. Dabei wird die Methode `getContent()` aufgerufen, die ein „Object“ zurückgibt, das den Inhalt der Resource darstellt.
  In der Implementierung nach [Sie05] wurde hier der Inhalt der Resource als Datei zurückgegeben. Die GUI erwartet den Inhalt aber als String. Deswegen musste die Klasse `T1_EdgeInline`, die für die Rekonstruktion der XML-Daten aus der Datenbank zuständig ist, so umgeschrieben werden, dass der Inhalt in einen String und nicht in eine Datei geschrieben wird.

- Ein ähnliches Problem gab es in der Methode `setContent(Object o)` von `XMLResource`. Hier ist der Eingabeparameter, der den zu setzenden Inhalt darstellt, ebenfalls vom Typ „Object“.
  Dieser Parameter wurde in der bisherigen Implementierung nach [Sie05] zu einem String konvertiert, der den Pfad zu der Datei enthält, die den Inhalt darstellt.
  Der GUI-Client geht aber davon aus, dass der zu setzende Inhalt bereits in dem „Object o“ (als String) enthalten ist.
  Die Interpretation des „Objects“ als String, der den Pfad zur Datei darstellt, erscheint nicht sehr schlüssig. Es sollte allerdings möglich sein, den Inhalt auch aus einer Datei in die Resource zu schreiben.
  Zur Lösung dieses Problems wurde eine Fallunterscheidung eingebaut, die je nach Art der Eingabe die richtige Verarbeitung anstößt. Die Eingabe kann dabei vom Typ String oder vom Typ FileInputStream sein.

- In der Methode `queryResource(String id, String query)` im `XPathQueryServiceImpl` wurden die Eingabeparameter vertauscht. Hier muss zuerst die Id der befragten Resource und dann die Query stehen. Dies wurde berichtigt.

Des Weiteren wurde festgestellt, dass der GUI-Client beim Importieren einer Resource eine andere Vorgehensweise hat als erwartet: Die Resource wird erzeugt, der zu setzende Inhalt in einen DOM-Knoten eingelesen und dessen Inhalt mit `setContentAsDOM(Node n)` der Resource zugeordnet.
Das heißt, der Inhalt wird zuerst geparst, als DOM-Baum dargestellt, dann wird der DOM-Baum nochmal geparst und der Inhalt der Resource gesetzt. Dieses zweimalige Parsen ist unnötig. Außerdem führt das Darstellen des Inhalts als DOM-Baum zu Speicherengpässen. Das Importieren einer 1MB-Datei führte bereits zum Absturz des GUI-Clients.
Besser wäre es, wenn der Inhalt der Resource mit `setContent(Object o)` gesetzt würde, da hier nur einmal der Inhalt mit SAX geparst und der Inhalt der Resource direkt gesetzt wird.
Dies kann nur durch eine Änderung des Source-Codes des GUI-Clients geschehen. Die GUI bietet die Möglichkeit, ihre Funktionalität zu erweitern [Tit06], indem eine Klasse geschrieben wird, welche die `XMLdbUtilImpl`-Klasse erweitert. Die `XMLdbUtilImpl`-Klasse beinhaltet z.B. die Operationen zum Erzeugen

und Löschen von Resourcen. In der abgeleiteten Klasse können beliebige Methoden überschrieben werden. Damit die GUI die neue Klasse erkennt, muss diese in der Konfigurationsdatei registriert werden.
Um das Importieren von Resourcen effizienter zu gestalten, wurde in der neuen Version der Middleware eine Klasse `XMLDBUtilImpl_Extend` hinzugefügt, die von der Klasse `XMLdbUtilImpl` erbt und die Methode `Resource createXMLResource(Collection col, String docId, InputStream is)` überschreibt. In deren Implementierung wird der Inhalt der Resource direkt mit `setContent(Object o)` gesetzt und somit das zweimalige Parsen und damit verbundene Einlesen des Inhalts in einen DOM-Knoten verhindert.
Im Folgenden sollen einige Screenshots die XML:DB-GUI zeigen.

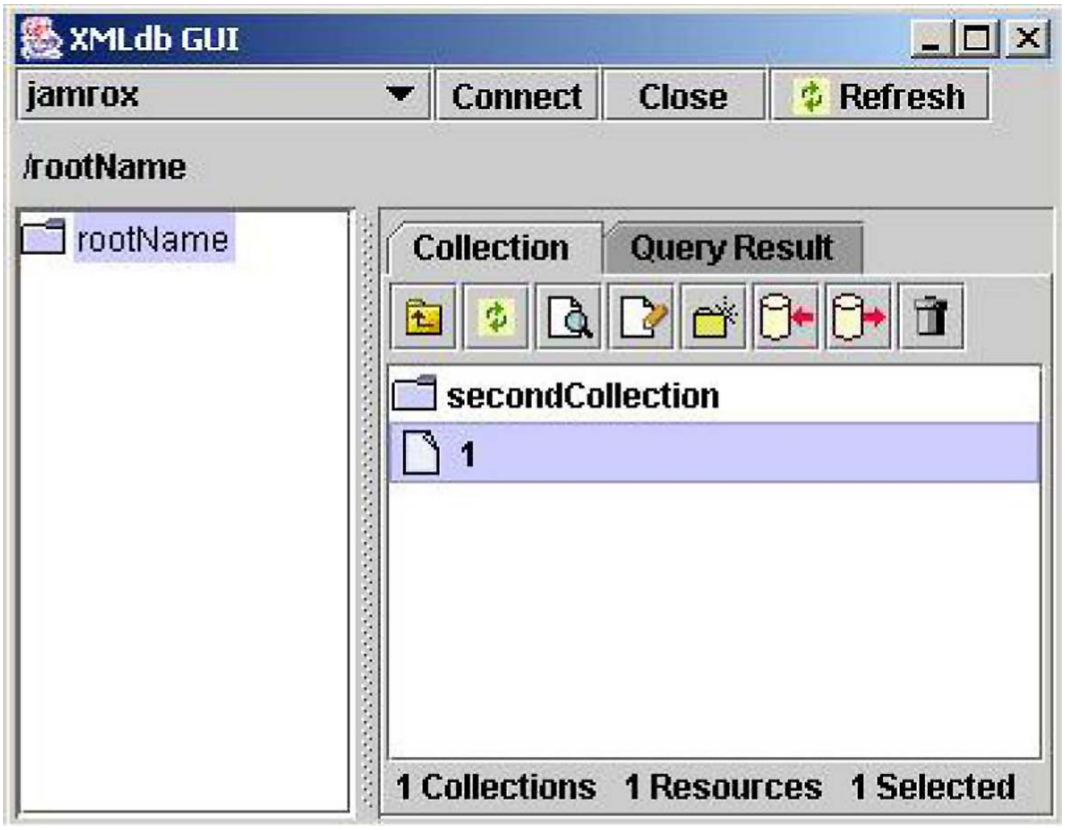


Abbildung 11: Die XML:DB-GUI

In Abbildung 11 wird die Maske der XML:DB-GUI gezeigt, die sich nach Starten der GUI öffnet. Links oben kann man sehen, dass sie mit der „jamrox“-Konfiguration gestartet wurde. Auf der linken Seite wird die root-Collection angezeigt und auf der rechten Seite deren Inhalt.
Sie enthält eine weitere Collection namens „secondCollection“ und eine Resource mit der Id 1. Da diese Resource markiert wurde, steht unten rechts „1 selected“.
Mit einem Doppelklick auf die Resource, öffnet sich ein neues Fenster, das den Inhalt der Resource anzeigt. Dies wird in Abbildung 12 für die Resource 1 gezeigt.

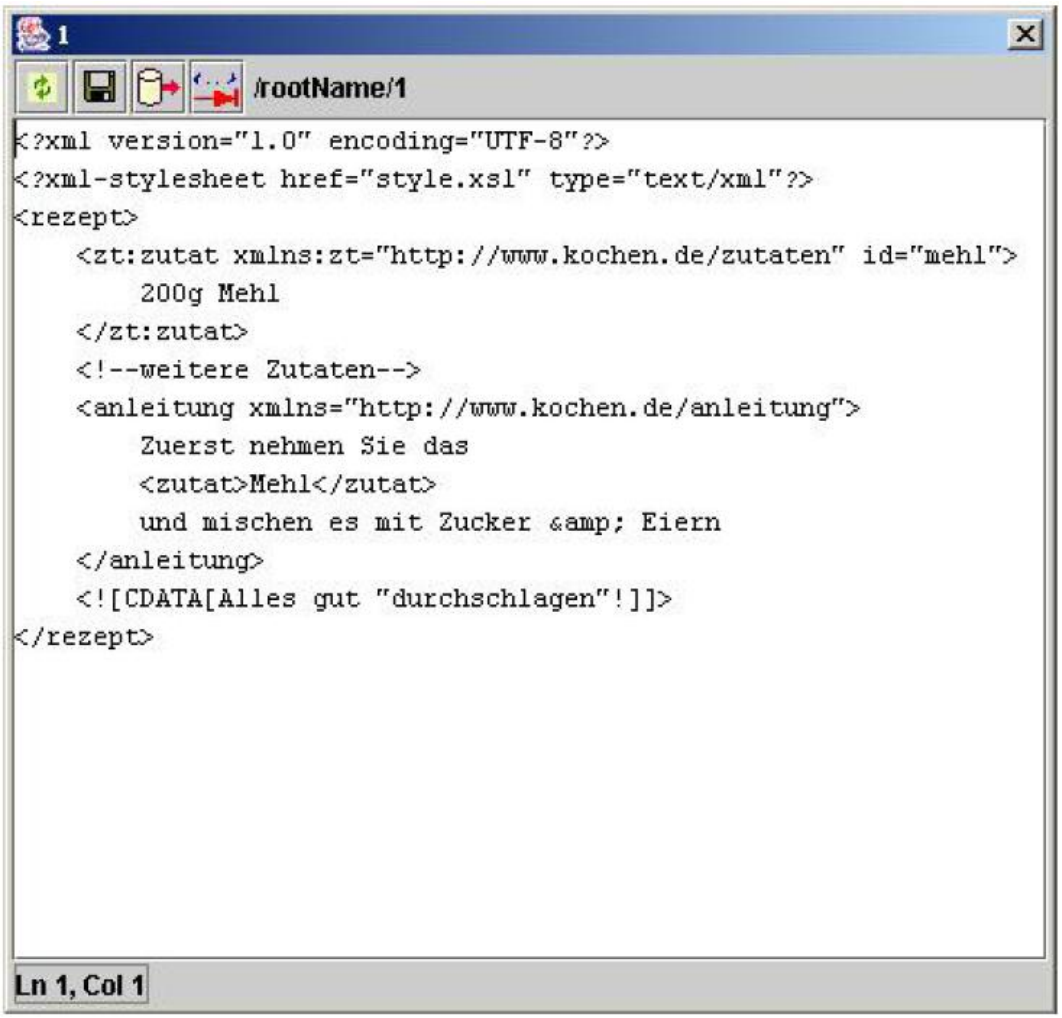


Abbildung 12: Inhalt der Resource mit id „1“

Um eine Query auf einer Resource ausführen zu können, muss diese im eben gezeigten Startfenster markiert werden und dann der dritte Button von links in der Menüleiste über dem rechten Fenster angeklickt werden.
Es öffnet sich ein neues Dialogfenster.

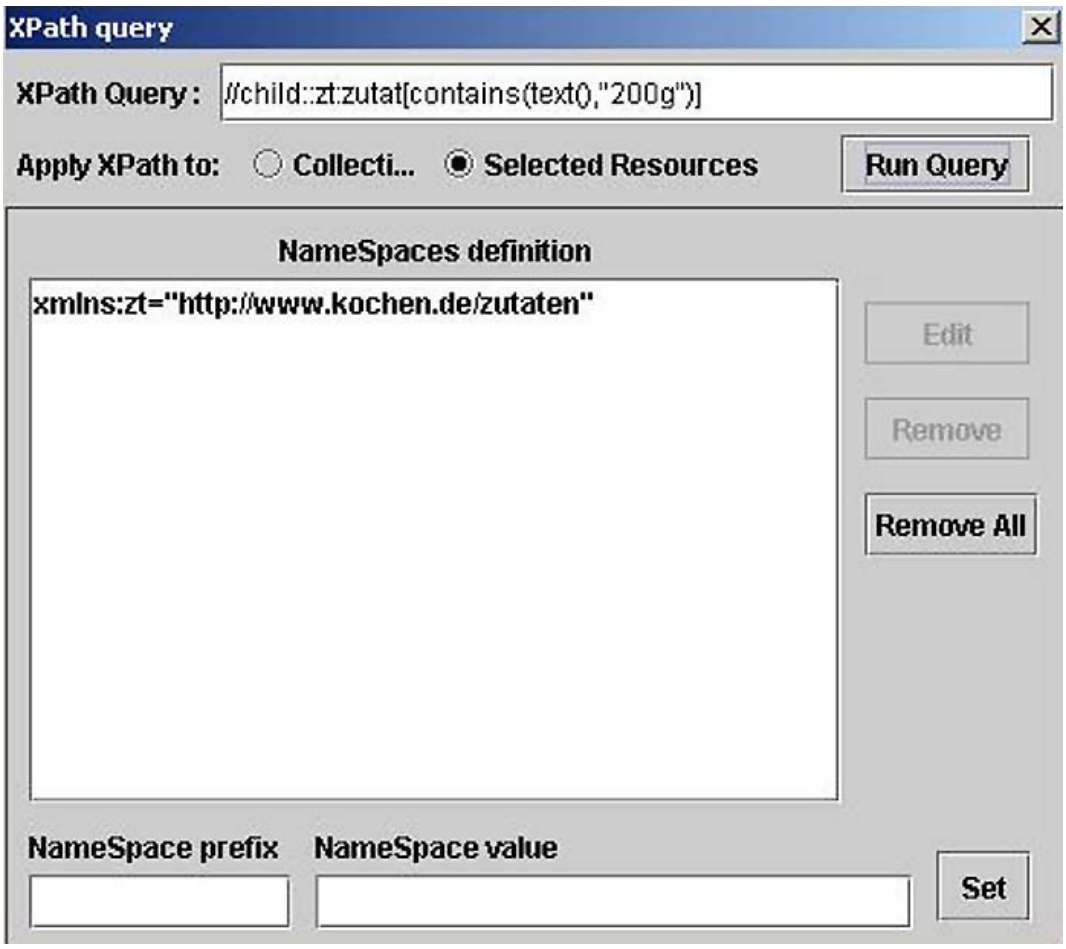


Abbildung 13: Dialogfenster für XPath-Queries

Hier werden die Namensraumabbildungen deklariert und eine Query eingegeben. Außerdem kann angegeben werden, ob die Query auf die ausgewählte Resource oder der Collection ausgeführt werden soll. Beispielhaft wird dieses Fenster in Abbildung 13 dargestellt.
Das Ergebnis der Query wird in einer Liste von Resourcen im Reiter „QueryResult“ dargestellt, denen eine eindeutige Id zugeordnet wurde. Die Query aus Abbildung 13 hat nur eine Resource als Ergebnis. Dies ist in Abbildung 14 dargestellt.
Das Fenster in Abbildung 15 wird bei Doppelklick auf die Resource geöffnet und zeigt den Inhalt der Ergebnis-Resource an.

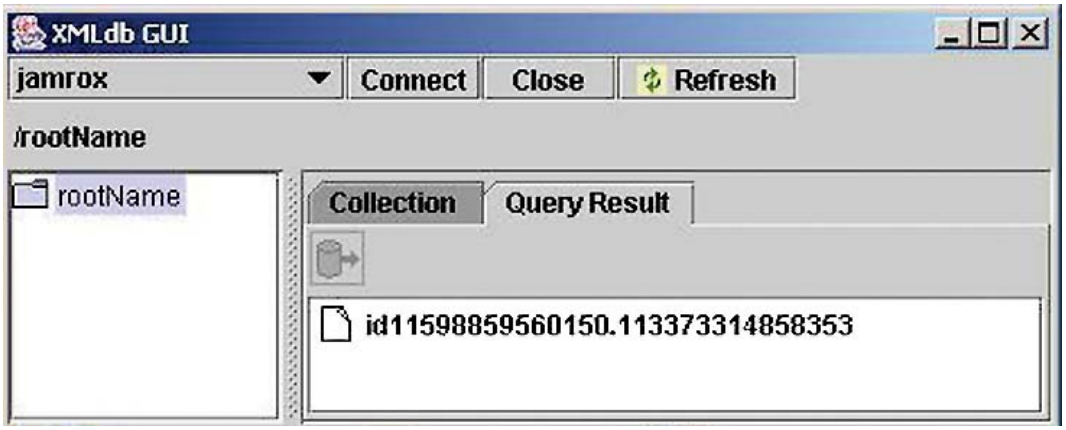


Abbildung 14: Anzeige der Ergebnisse als Liste

Abbildung 15: Inhalt des Ergebnisses

5.2 Performance

5.2.1 Steigerung der Geschwindigkeit beim Einfügen von Dokumenten

Das Hinzufügen des TransactionServices hat die Geschwindigkeit beim Importieren eines Dokuments in die Datenbank verbessert.
In Kapitel 6 werden die Messungen der Zeit beim Einfügen von Dokumenten ohne TransactionService und mit TransactionService verglichen.

5.2.2 Steigerung der Geschwindigkeit bei Anfragen

Die Geschwindigkeit, mit der Anfragen bearbeitet werden, hängt sehr stark von der Vorgehensweise der Jaxen XPath Engine bei der Bearbeitung der Anfragen ab. Hierbei gibt es in Jaxen einen objektmodell-unabhängigen Teil, der das Parsen des XPath-Ausdrucks übernimmt und einen objektmodell-abhängigen Teil, der für die Navigation auf dem gewählten Objektmodell zuständig ist [Cod06b].
Die Navigation wird durch den `JamroxNavigator` realisiert, der für jede XPath-Achse (z.B. die child-Achse) die entsprechende DOM-Anfrage (z.B. getFirstChild()) stellt. Durch das Bereitstellen der Daten aus der Datenbank als DOM kann der `JamroxNavigator` auf die Daten zugreifen. Diese Vorgehensweise bei der Navigation soll nicht grundlegend verändert werden. Daher müssen andere Ansatzpunkte gefunden werden, um die Anfragebearbeitung zu beschleunigen. Hierbei war die erste Idee, schon die Eingabe, d.h. den XPath-Ausdruck, in einer Form bereitzustellen, mit der die Verarbeitung schneller vonstatten gehen kann. Nach eingehender Beschäftigung mit der XPath-Spezifikation und Suche nach einer Beschreibung einer XPath-Normalform musste festgestellt werden, dass es keine Möglichkeit gibt, einen XPath-Ausdruck in eine Normalform umzuschreiben.
In einigen Veröffentlichungen (z.B.in [Gen04]) wird zwar eine Vorgehensweise zur schnelleren Bearbeitung von XPath-Ausdrücken beschrieben, jedoch wird hier immer Bezug auf die Arbeitsweise der XPath Engine genommen. Die Arbeitsweise der XPathEngine soll im Rahmen dieser Arbeit nicht grundlegend verändert werden.
Eine Optimierung der Performance bei der Anfragebearbeitung durch Umformen des XPath-Ausdrucks ist folglich nicht möglich.
Ein anderer Teil der Anfragebearbeitung, der von uns gestaltet wird, ist der Zugriff auf die Datenbank. Hier könnten die SQL-Anfragen weiter optimiert werden.
Die „prepared Statements“ wurden dahingehend überprüft und es wurde festgestellt, dass die Statements bereits in einer optimalen Form vorliegen und nicht weiter vereinfacht werden können.
Als Beispiel soll das prepared Statement zum Auswählen des ersten Kindes eines Elements gezeigt werden:

```
select * from tableName where parent=? and ord='1'
and type!='Attribute';
```

Ein eher spezieller Ansatzpunkt, die Anfragebearbeitung zu verbessern, ist das

„full text indexing“ zu ermöglichen. Wie bereits in Kapitel 4.2.2 erwähnt, sollte dies unabhängig von der Datenbank erfolgen, so dass die Datenbank ohne Probleme ausgetauscht werden könnte.
Diese Möglichkeit bietet die „Information Retrieval“-Bibliothek Lucene [Apa06a], mit der Indizierung und Suchfähigkeiten einer Anwendung hinzugefügt werden können [Gos05]. Lucene kann alle Arten von Datenquellen, die in Textformat umgewandelt werden können, indizieren.
Lucene führt den Prozess der Indizierung in drei Teilen durch wie in Abbildung 16 dargestellt ist.

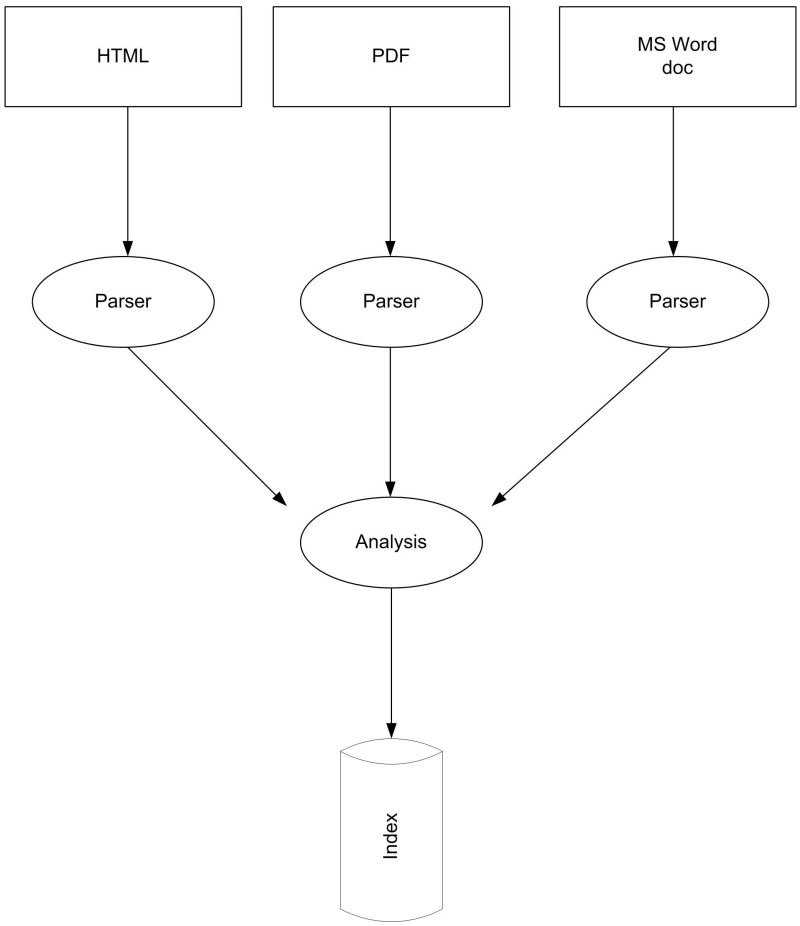


Abbildung 16: Indizierungsprozess mit Lucene [Gos05]

Zuerst muss die Datenquelle geparst werden, um den zu indizierenden Text zu extrahieren. Für jeden Text wird eine Instanz der Lucene-Klasse `Document` erzeugt.
Ein `Document` repräsentiert eine Sammlung von Feldern, in denen die eigentlichen Daten des Dokuments oder Meta-Daten des Dokuments stehen können. Lucene stellt diese Felder mit der Klasse `Field` dar, wobei jedem Feld ein Name, ein Wert und drei boolsche Attribute zugeordnet wird. Die boolschen Attribute beschreiben, wie der Wert des Feldes verarbeitet werden soll und haben folgende Bedeutung, wenn sie auf `true` gesetzt sind:

- `indexed`: das Feld wird indexiert, kann also von Lucene durchsucht werden.
- `tokenized`: das Feld wird von dem später auszuwählenden Analyzer „tokenisiert“, d.h. in Einzelbestandteile zerlegt und Teile herausgefiltert.
- `stored`: der Feldinhalt wird im Index gespeichert. Man kann im Allgemeinen aus einem indexierten Feld nicht den gesamten Feldinhalt rekonstruieren. [The04].

Nach der Erstellung der `Document`-Instanz(en) wird der Text analysiert, was z.B. bedeutet, dass er „tokenisiert“ wird, bedeutungslose Tokens entfernt werden usw.. Dafür bietet Lucene verschiedene Arten von `Analyzern`.
Die Klasse `IndexWriter` erzeugt einen Index. Dabei muss man angegeben, wo der Index gespeichert werden soll. Dafür gibt es die Möglichkeit, den Index im Dateisystem in einem Verzeichnis zu speichern (`FSDirectory`) oder den Index im Speicher zu erstellen (`RAMDirectory`).
Nachdem der Index erzeugt wurde, werden die erstellten `Documents` dem Index hinzugefügt.
Soll im Index nach einem String gesucht werden, muss eine `Query` erstellt werden, die als Eingabe einen `Term` benötigt. Dieser `Term` kann analog zum `Field`-Objekt gesehen werden, da er auch aus dem Namen des zu durchsuchenden Feldes und dem Wert, nach dem gesucht werden soll, besteht. Der Suchwert kann auch Wildcards, boolsche Werte usw. enthalten.
Die Ergebnisse der Suche werden in einer Instanz der Klasse `Hits` gespeichert, die einen Container mit Pointern auf die Suchergebnisse darstellt. [Gos05]
In der neuen Version der Middleware wurde das „full text indexing“ mit Lucene folgendermaßen umgesetzt:
Es wurde ein neues Paket `jamrox.lucene` erstellt, das eine Klasse `DB_Indexer` und eine Klasse `DB_Searcher` enthält.
Um Daten aus einer Datenbank zu indizieren, muss für jedes Tupel ein `Document` erzeugt werden [Apa06b]. In der Klasse `DB_Indexer` wird jedes Tupel vom Typ `Text` extrahiert und für jeden Textknoten ein `Document` erzeugt.
Diesem `Document` wird ein Feld `content` zugeordnet, welches den Textwert enthält und ein Feld `id`, das die Datenbank-Id des Textknotens enthält. In der Methode `indexNode(IndexWriter writer)` wird der Textknoten dem Index hinzugefügt.
Der gerade beschriebene Prozess des Indizierens wird in der Middleware nach dem Erzeugen einer `XMLResource` in der Methode `storeResource` angestoßen.
Um zu erklären, wie der Suchprozess in die Query Engine Jaxen integriert wurde, soll an einem Beispiel gezeigt werden, wie Jaxen eine „contains“-Anfrage bearbeitet. Eine Beispiel-XPath-Anfrage sieht so aus:

```
//child:node()[contains(text(),'suchstring']
```

Diese Query sucht alle Elemente, deren Text-Kindknoten den String “suchstring“ enthalten. Bei dieser Anfrage wird Jaxen für jeden Knoten im Dokument die child-Achse aufrufen und so die Kindknoten jedes Elements erhalten. Für diese Knoten ruft Jaxen dann die „Contains“-Funktion auf.
Jaxen stellt für jede Funktion eine eigene Klasse zur Verfügung, so auch für die

„Contains“-Funktion. Die Methode `evaluate` der Klasse `ContainsFunction` wird für jeden Knoten, der durch die Achse ausgewählt wurde, aufgerufen. In unserem Beispiel würde für jeden Kindknoten aller Elemente die `evaluate`-Methode aufgerufen.
Diese Methode überprüft, ob der Kindknoten ein Textknoten ist und wenn ja, ob der „suchstring“ im Wert des Textknotens enthalten ist. Dieses Durchsuchen erfolgt mit der Java-Methode `indexOf(String suchstring)`, welche die Position des Suchstrings zurückgibt. Findet diese Methode ein Vorkommen des Suchstrings, wird das Vaterelement des gerade untersuchten Knotens in die Ergebnismenge aufgenommen, sonst nicht.
In der neuen Version der Middleware wurde nun in die `Contains`-Klasse eine neue Methode `evaluateIndex` eingefügt, welche die `evaluate`-Methode ersetzen soll und dabei Folgendes tut:
Beim ersten Aufruf der Methode wird die neue Klasse `DB_Searcher` instanziiert und damit der „suchstring“ im Index gesucht. In `DB_Searcher` wird die Suchanfrage so gestellt, dass der „suchstring“ auch Teil eines anderen Wortes sein könnte. Als Ergebnis werden die Id's der Knoten, in denen der „suchstring“ vorkam, in einer der `Contains`-Klasse hinzugefügten Array-Variable gespeichert.
Nun kann bei diesem Aufruf und bei allen folgenden die Id des gerade betrachteten Knotens mit den Id's aus den Suchergebnissen verglichen werden, um festzustellen, ob der „suchstring“ im Wert des gerade betrachteten Knoten enthalten ist. Dazu muss nicht mehr der ganze Wert des Knotens durchsucht werden. Ist die Id des überprüften Knotens in den Suchergebnissen enthalten, wird der Vaterknoten in die Ergebnismenge aufgenommen, sonst nicht.
Die Methode `evaluateIndex` wird von der `evaluate`-Methode aufgerufen, wenn diese mit einem Textknoten als Parameter aufgerufen wurde, da wir nur diese indiziert haben.
Der Nachteil bei der gerade beschriebenen Funktionsweise der `evaluateIndex`-Methode ist, dass man, obwohl man mit Hilfe des Lucene-Indexes schon direkt sagen könnte, in welchen Textknoten der „suchstring“ enthalten ist, trotzdem noch alle von der Achse ausgewählten Knoten betrachten muss. Dies ist allerdings nicht anders möglich, wenn die Navigation der Jaxen XPath Engine nicht grundlegend verändert werden soll.
Eine andere Möglichkeit, Lucene einzusetzen, um die Anfragebearbeitung zu beschleunigen, ist folgende: In XPath können in Funktionen wie z.B. „contains“, „startsWith“ o.ä. nicht nur Textknoten als Parameter eingegeben werden, sondern auch Elementknoten. Dabei wird der „Zeichenkettenwert“ des Elementknotens als Vergleichskriterium benutzt.
Der „Zeichenkettenwert“ eines Elements ist die Verkettung der Zeichenkettenwerte aller Element- und Text-Kindknoten [W3C06c]. Der Zeichenkettenwert eines Textknotens entspricht dem Wert des Textknotens.
Wenn in Jaxen der Zeichenkettenwert eines Elements berechnet wird, müssen alle Kindknoten des Elements in der Datenbank gesucht werden, um wiederum deren Zeichenkettenwert zu berechnen. Das heißt, dass zum Berechnen des Zeichenkettenwerts eines Elements unter Umständen einige Datenbankzugriffe nötig sind.
Mit Lucene könnte man beim Einlesen eines Dokuments für jedes Element ein

`Document` anlegen und indizieren. Bei diesem `Document` könnte ein Feld mit der Id des Elements befüllt werden und das andere mit dem Zeichenkettenwert dieses Elements.
Jaxen müsste so beim Berechnen des Zeichenkettenwerts nicht mehrmals auf die Datenbank greifen, sondern es müsste nur noch ein Zugriff auf den Index erfolgen.
Das Problem bei dieser Idee ist allerdings, dass der Index viel Speicherplatz bräuchte, da die Zeichenkettenwerte von Elementen eventuell sehr lange werden können. So entspricht beispielsweise der Zeichenkettenwert des Wurzelknotens der Verkettung von allen in dem Dokument vorkommenden Textknoten. Aus diesem Grund wurde diese Idee in der neuen Middleware nicht realisiert.

5.3 Zuverlässigkeit

5.3.1 Testen

In Kapitel 5.1.7 wurde beschrieben, wie die Implementierung der Middleware verändert wurde, um eine Konformität zur XML:DB-API zu erreichen. Diese Veränderungen erforderten eine Anpassung der JUnit-Tests, die bereits für alle XML:DB-API-Klassen vorhanden sind.
Darüber hinaus muss die in der neuen Version der Middleware hinzukommende Funktionalität (z.B. Unterstützung von DOM und SAX) mit neuen Testfällen abgedeckt werden.
Zur Realisierung des ersten Punkts wurden folgende Änderungen an den Testfällen durchgeführt:

- In der Klasse `DatabaseTest` wurde die richtige URI zur Datenbank eingefügt, um die Verbindung aufbauen zu können.
- In der Klasse `XMLResourceTest` wurde die Methode `testSetContent` angepasst, so dass nun beide möglichen Eingabetypen (File oder String) getestet werden. Die Methode `testGetContent` wurde verändert, so dass nun der Rückgabetyp einem String entspricht.
- In allen Testklassen wurde beim Erzeugen einer Resource der Typ von „XML_RESOURCE“ in „XMLResource“ umgeändert. Außerdem wurde in allen Klassen bei der Erzeugung einer XPath-Query die Reihenfolge der Parameter (erst Resource, dann Query) berichtigt.

Zum Testen der neuen Funktionalitäten wurde eine Testdatei erstellt, welche alle XML-Konstrukte enthält, die nun in der neuen Version der Middleware unterstützt werden. Diese Testdatei „rezept.xml“ wurde bereits in Kapitel 4.1.2 vorgestellt (Abbildung 6).
Sie wurde zusätzlich in zwei leicht abgeänderten Versionen erstellt. Dies war für die DOM- und SAX-Methoden notwendig, da SAX ohne LexicalHandler (siehe Kapitel 5.1.4) keine Kommentare, CDATA-Abschnitte usw. unterstützt und der DOM-Parser normalerweise Entity-Referenzen nicht als separate Knoten darstellt, sondern in den zugehörigen Wert umwandelt [Har04]. Die abgewandelten Versionen von „rezept.xml“ enthalten keine der nicht unterstützten Knoten.

Es wurden folgende Testmethoden erweitert bzw. konnten erst implementiert werden:

- In der Klasse `CollectionTest` wurde zur Methode `testGetService()` der TransactionService und der XUpdateQueryService hinzugefügt.

- In der Klasse `XMLResourceTest` wurden die Methoden `testGetContentAsSAX()`, `testSetContentAsSAX()`, `testGetContentAsDOM()` und `testSetContentAsDOM()` hinzugefügt.
  Für die DOM-Testmethoden ist es nötig, den Inhalt eines DOM-Knotens mit dem aus einer Datei bzw. aus einer Resource vergleichen zu können. Deswegen wurde der Testklasse die Methode `getNodeContentAsString()` hinzugefügt, die einen DOM-Baum traversiert und als String zurückgibt. Um zu testen, ob mit `getContentAsSAX(Handler handler)` der Inhalt einer Resource richtig in Form von SAX-Events ausgegeben wird, wurde eine innere Klasse `SaxAsString` eingefügt, die einen Handler implementiert, welcher die SAX-Events entgegennimmt und die XML-Knoten als Strings ausgibt.

- In der Klasse `XPathQueryServiceTest` wurden in der Methode `testQueryResource` die Node Tests `comment()` und `processing-instruction()` als Testfälle ergänzt.
  Außerdem wurde das Testen der nun unterstützten Namespace-Achse und der Node Set Functions `string local-name(node-set?)`, `string namespace-uri(node-set?)` und `string name(node-set?)` hinzugefügt.
  Der Zugriff auf die „neuen“ Knoten (z.B. Comment-Knoten), die nun von der Middleware unterstützt werden, wurde mit XPathQueries getestet, die die entsprechenden Knoten als Ergebnis zurückliefern.
  Implementiert werden konnten die Testmethoden `testGetNamespace`, `testSetNamespace`, `testClearNamespaces` und `testRemoveNamespace`. Dabei wird nur überprüft, ob die gesetzten Namensraumabbildungen in der internen Namensraumtabelle eingefügt bzw. gelöscht wurden. Die eigentliche Funktionalität, die aus dem Setzen von Namensraumabbildungen folgt, wird in `testQueryResource` getestet.
  Hier werden einige XPath-Queries getestet, in denen Elemente vorkommen, die in einem Namensraum liegen. Ohne Setzen der entsprechenden Namensraumabbildungen dürfen diese Queries nur zu einem leeren Ergebnis führen.

- In der Klasse `JamroxDOMTest` wurden Testfälle hinzugefügt, so dass nun auch alle „neuen“ DOM-Knoten von den verschiedenen DOM-Methoden zurückgegeben werden.

- Neu hinzugefügt wurde die Klasse `TransactionServiceTest`, die alle Transaktions-Methoden testet. Diese Testklasse wurde der Klasse `AllTests` und damit der TestSuite für alle XML:DB-API-Klassen hinzugefügt.

In allen schon vorhandenen Testfällen wurden Kommentare und Fehlermeldungen überarbeitet, ergänzt und in Form und Sprache vereinheitlicht.
Nicht getestet wurde der XUpdateQueryService, da die Spezifikation der Update-Sprache XUpdate widersprüchlich und unvollständig ist (siehe Kapitel 5.1.5). Es gibt folglich keine eindeutigen Vorgaben, anhand derer die Testfälle definiert werden könnten.
Beim Testen des TransactionService fiel auf, dass das Anlegen des Indexes zum Abbruch der Transaktion führte. Dies hatte folgende Ursache: Es gab nur **einen** Index, der jeweils für die aktuell erzeugte Tabelle angelegt wurde.
Das heißt, wenn eine Tabelle angelegt wurde, wurde für diese der Index mit dem Namen „query_parent“ erzeugt. Sobald eine neue Resource angelegt wurde, wurde eine Exception ausgelöst, da der Index „query_parent“ bereits existierte. In der Fehlerbehandlung wurde der Index gelöscht und für die neue Tabelle neu angelegt. Dieses Vorgehen hatte zur Folge, dass die Transaktion abgebrochen wurde, da während einer Transaktion keine Exception geworfen werden darf.
Das Anlegen des Indexes wurde geändert, für jede Tabelle wird ein neuer Index angelegt. Dieser Index besitzt einen eindeutigen Namen.

5.3.2 Code-Analyse

Durch das Anbinden des GUI-Clients und die damit verbundene intensive Beschäftigung mit der XML:DB-API und deren Implementierung in der Middleware fielen einige Stellen auf, an denen Optimierungen des Codes sinnvoll wären. Im Folgenden soll beispielhaft für die Klasse `XMLResource` erläutert werden, was in der neuen Version der Middleware verbessert wurde:

- Im Interface `XMLResource` gibt es zwei Attribute `id` und `docId`, wobei `id` die eindeutige Id der Resource darstellt und `docId` die Id der befragten „parent“-Resource, falls die Resource das Ergebnis einer Query ist. Ist die Resource nicht Ergebnis einer Query, ist der Wert von `docId` gleich `id` [XML06e].
  In der Version nach [Sie05] gibt es für die Ergebnisse einer Query die spezielle Implementierung `QueryResourceImpl`, so dass in der Klasse `XMLResourceImpl` die Unterscheidung zwischen den beiden Id's nicht notwendig ist.
  Es müssen im Konstruktor folglich nicht mehr extra beide Id's als Parameter mitgegeben werden, da beide immer gleich sind. Dies wurde entsprechend vereinfacht. Die Unterscheidung zwischen den Id's wurde in der Version nach [Sie05] in `QueryResourceImpl` nicht umgesetzt. Hier ist sie aber notwendig. Dies wurde in der neuen Version entsprechend der API-Spezifikation geändert.

- Bei der Reproduktion eines XML-Dokuments aus der Datenbank wurde zweimal nach dem Wert der Spalte „tableName“ für eine Resource-Id gesucht (einmal in der Methode `getContent()` und das zweite Mal im Konstruktor der Klasse `T1_EdgeInline`).
  Die zweite Suche in der Datenbank wurde entfernt und der Wert des Tabellennamens im Konstruktor von `T1_EdgeInline` übergeben.

Weitere Veränderungen am Quellcode werden im Folgenden beschrieben:
Um die Formatierung des Quellcodes zu vereinheitlichen, wurde die Format-Funktion von Eclipse genutzt. Dabei wurde das voreingestellte „Eclipse 2.1“-Profil verwendet.
Um den Quellcode weiter zu bereinigen, wurden zwei Werkzeuge zur Code-Analyse und ein Debugger eingesetzt.
Im Folgenden wird erläutert, wie der Quellcode mit Hilfe der Analyse-Ergebnisse verändert wurde.
Zuerst wurde das Checkstyle-Tool benutzt. Bei manchen Regeln bietet Checkstyle an, die Verletzung der Regel mit einem Quickfix (z.B. Entfernen von leeren Anweisungen) zu beheben, bei den restlichen muss der Code manuell angepasst werden.
Die Analyse wird bei jedem Komplilier-Vorgang angestoßen und reagiert somit sofort (bei eingeschaltetem Auto-build) auf Änderungen am Quellcode. Die Menge der Regeln, die für die Analyse benutzt werden, kann beliebig konfiguriert werden.
Die Checkstyle-Konfiguration Sun Checks wird bereits mitgeliefert. Sie entspricht den von SUN definierten Code-Konventionen [Sun06b] und beinhaltet somit die wichtigsten anerkannten Code-Standards. In der Beschreibung der Code-Konventionen [Sun06c] wird die Dringlichkeit, diese einzuhalten, folgendermaßen begründet:

- 80 % der Kosten einer Software während ihrer Laufzeit entstehen durch die Wartung.
- Fast keine Software wird durch ihren Autor gewartet.
- Code-Konventionen verbessern die Lesbarkeit von Software, so dass Entwickler neuen Code schneller und gründlich verstehen können.
- Wenn der Source Code als Produkt ausgeliefert werden soll, muss sichergestellt werden, dass er gut verpackt und sauber ist, wie jedes andere Produkt, das erstellt wird.

Checkstyle stellt außerdem eine leicht veränderte Version von „Sun Checks“ bereit, welche besser mit den Default-Codeformatierungs-Einstellungen von Eclipse harmoniert. Sie wurde zur Analyse des Quellcodes der Middleware benutzt.
Die Regeln werden in Checkstyle thematisch zusammenhängenden Blöcken zugeordnet. In der folgenden Aufzählung werden die in der Konfiguration enthaltenen Blöcke erklärt und dargestellt, wie der Quellcode aufgrund der jeweiligen Regeln angepasst wurde:

- `Javadoc Comments`: Diese Regel checkt, ob alle Javadoc-Kommentare vorhanden sind und ob diese „wohlgeformt“ sind, d.h. alle „@“- Tags vorhanden sind, nach dem ersten Satz ein Punkt steht usw..
  Aufgrund der Hinweise diesbezüglich wurden im Quellcode fehlende Javadocs ergänzt und syntaktisch verbessert.
- `Naming Conventions`: Für jedes Modul (z.B. Klasse, Methode, Konstante) gibt es einen regulären Ausdruck, der das Format von Bezeichnern

dieser Module vorgibt.
Auf Basis dieser Vorgaben mussten einige Module in der Middleware umbenannt werden, wie z.B. Klassen, deren Bezeichner Unterstriche enthalten (z.B. `T1_EdgeInline`) oder abstrakte Klassen. Dabei musste bei ersterem der Unterstrich entfernt werden und bei letzterem das Präfix `Abstract` eingefügt werden.

- `Imports`: Importe mit „*“ sollten vermieden werden, doppelte und überflüssige Importe entfernt werden.
  Alle Import-Befehle in der Middleware wurden dementsprechend angepasst.

- `Size Violations`: Hier werden Regeln festgelegt, die die Länge von Zeilen, Methoden, Klassen usw. beschränken.
  Die Middleware wurde bezüglich dieser Regeln nicht angepasst, da z.B. die Zeilenlänge bereits durch die Formatierungs-Funktion von Eclipse vereinheitlicht wurde.

- `Whitespace`: Diese Regel sichert zu, dass an bestimmten Stellen (z.B. nach einem Token wie „,“ oder „;“) ein Leerzeichen steht bzw. an bestimmten Stellen kein Leerzeichen steht.
  Diese Regeln wurden bereits durch die Formatierungs-Funktion von Eclipse umgesetzt.

- `Modifiers`: Modifier werden daraufhin geprüft, ob die Reihenfolge dieser in der Middleware bei der Deklarierung den Vorschlägen der „Java Language Specification“ entspricht (`public, protected, private,...`).
  Die Deklarationen in der Middleware wurden auf Basis dieser Vorgaben neu sortiert.

- `Blocks`: Es wird nach leeren Blöcken und nach Blöcken gesucht, bei denen die geschweiften Klammern fehlen.
  In unserem Quellcode wurden auf Basis dieser Ergebnisse leere Catch-Blöcke ersetzt und zur besseren Lesbarkeit überall geschweifte Klammern eingesetzt.

- `Coding Problems`: In diesem Block wird überprüft, ob es redundante Initialisierungen (Zuweisen von „0“ oder „null“) oder Klammern (z.B. beim Casten) gibt, ob leere Anweisungen vorhanden sind und ob boolsche und return-Ausdrücke vereinfacht werden können.
  Außerdem wird gecheckt, ob der Zugriff auf Instanzvariablen nur über getter- und setter-Methoden geregelt wird und ob häufig vorkommende Zahlen und Strings in Variablen ausgelagert werden könnten.
  So konnten im Quellcode alle Redundanzen und zu komplizierte Ausdrücke entfernt werden.
  In der Klasse `JamroxSqlConnection` gab es in der Version nach [Sie05] keine getter-und setter-Methoden. Dies wurde geändert, so dass der Zugriff auf die „prepared Statements“ nun mit diesen erfolgt.
  Zusätzlich wurde in allen Jamrox-DOM-Klassen der Zugriff auf die Spalten in der Datenbank folgendermaßen geändert, so dass der Code leichter

wartbar ist:
In der Version nach [Sie05] sah ein Zugriff auf die Spalte *name* in der Datenbank so aus:
`name = rs.getString(2);`
Die Ordnung der Spalten wird in der neuen Version in Klassenvariablen der DOM-Klassen festgehalten, so dass in dem obigen Ausdruck anstatt der Zahl der Bezeichner der neuen Variable (in diesem Fall `sqlName`) steht. So müssen bei einer erneuten Änderung des EdgeInline-Schemas nur die Variablen an die neue Ordnung der Spalte angepasst werden.

Um noch andere Aspekte der Code-Analyse zu betrachten, wurde eine weitere Analyse mit der Entwicklungsumgebung IntelliJ durchgeführt.
Auch hier können die Aspekte, auf die der Code untersucht werden soll, konfiguriert werden. Bei der Analyse der Jamrox-Middleware wurde die Default-Konfiguration von IntelliJ benutzt, welche die wichtigsten Aspekte der Code-Analyse beinhaltet. Ergebnisse der Analyse und daraus folgende Änderungen am Code werden in der folgenden Aufzählung erläutert:

- `Probable Bugs`: Hier wird nach Ausdrücken gesucht, die zu einer `NullPointerException` führen können. Dabei wurden z.B. Ausdrücke gefunden, die zwei Ausdrücke mit `&` verknüpft enthalten. Dabei kann das Prüfen der zweiten Bedingnung zu einer `NullPointerException` führen, wenn das Auswerten des ersten Ausdrucks zum Ergebnis `false` führte. Deswegen muss bei solchen Ausdrücken das `&` durch ein `&&` ersetzt werden, damit die zweite Bedingung nicht mehr ausgewertet wird, wenn die erste `false` ist.
  Außerdem wird nach unbenutzten Zuweisungen gesucht. Diese finden sich oft bei der Deklaration von Variablen, die direkt initialisiert werden, dessen Wert aber nie gelesen wird. Diese Initialisierungen wurden entfernt.

- `Control Flow Issues`: Die Ergebnisse in diesem Bereich weisen auf redundante if-Abfragen hin, die in der Form `if(Bedingung) return true; else return false;` vorkommen und durch ein `return(Bedingung);` ersetzt werden können.

- `Class Structure`: Diese Regeln überprüfen die Struktur der Klassen z.B. auf Instanzvariablen, die auch lokal definiert werden könnten. Hier wurden entsprechende Anpassungen, soweit diese sinnvoll erschienen, durchgeführt.

Als Letztes wurde der Quellcode mit dem Tool „FindBug“ auf mögliche Bugs durchsucht. Dabei wurden folgende Meldungen erzeugt, die zu entsprechenden Änderungen führten:

- `Method may fail to close database resource`: Überall dort, wo JDBC-Statements erzeugt werden, sollten diese auch wieder geschlossen werden, sofern das Statement nicht als Rückgabewert dient oder weiter verwendet werden soll.
  Dies wurde im Quellcode bisher an keiner Stelle getan, was zu einer

schlechten Performance und Problemen bei der Kommunikation mit der Datenbank führen kann. In der neuen Version der Middleware wurde das Schließen der Statements in allen Klassen ergänzt, in denen auf die Datenbank zugegriffen wird.

- `Should be a static inner class`: In der Klasse `DOMHandler` wurde die innere Klasse `Iterator`, die dem Navigieren über den DOM-Baum dient, als innere Klasse erkannt, die statisch sein sollte. Dies wurde entsprechend geändert.

- `Load of known null value`: Hier wurde ein Fehler entdeckt, der durch falsches Setzen der Klammern eines if/else-Konstrukts entstand. Dadurch wurde ein `null`-Wert benutzt, was korrigiert wurde.

- `Nullcheck of value previously dereferenced`: Auch hier wurden die Klammern einer if/else-Abfrage falsch gesetzt. Dadurch fand die Abfrage, ob ein Objekt ungleich `null` ist, erst statt, nachdem bereits eine Methode dieses Objekts aufgerufen wurde. Die Klammern wurden aufgrund dieses Hinweises richtig gesetzt.

6 Performance-Tests

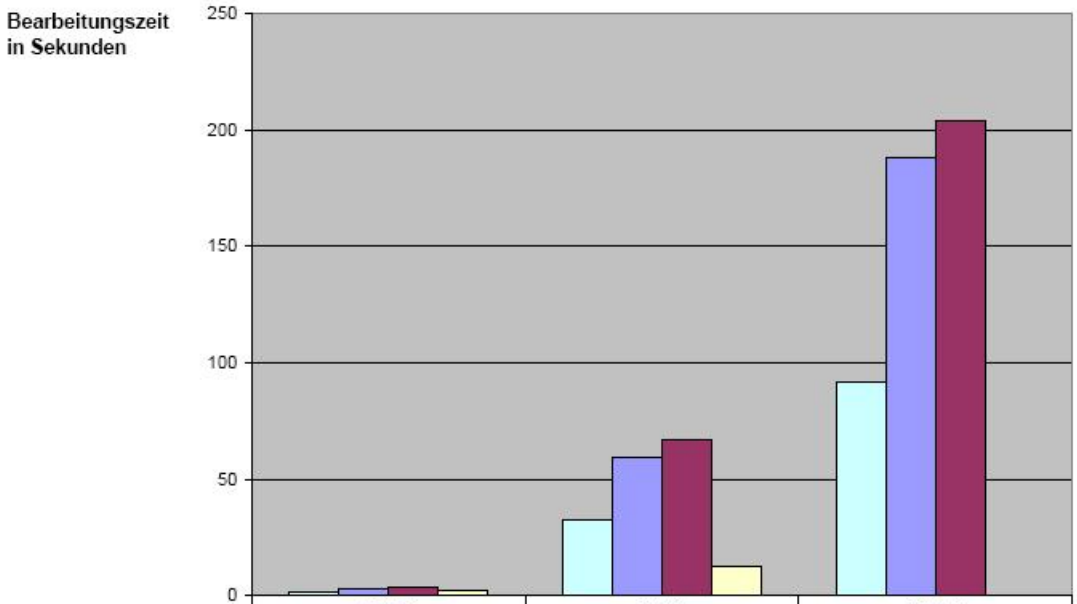
Damit man die neue Version der Middleware mit der Version nach [Sie05] und mit der nativen XML-Datenbank eXist vergleichen kann, wurden Benchmarks zusammengestellt, mit denen folgende Abläufe getestet werden:

- A1: Einlesen und Speichern von XML-Dokumenten
  Dabei wird das Einlesen von verschiedenen großen Dateien (1MB, 3 MB und 10MB) mit der Version nach [Sie05], der neuen Version der Middleware und eXist getestet.
  Das Einlesen wurde mit der neuen Version in zwei Varianten durchgeführt: einmal in einer Transaktion (mit Hilfe des neu hinzugefügten TransactionServices) und einmal ohne Transaktion, d.h. mit eingeschaltetem AutoCommit.

- A2: Ausführen eines Updates auf einer Resource mit Hilfe des XUpdateQueryService
  Hier wird die Geschwindigkeit beim Einlesen und Ausführen einer XUpdate-Datei mit neuer Version der Middleware mit der von eXist verglichen.

- A3: Ausführen einer „contains“-Query
  Das Ausführen einer XPath-Anfrage mit „contains“ wird mit der neuen Version von Jamrox ohne full text indexing, mit der neuen Version von Jamrox mit full text indexing und mit eXist durchgeführt.
  Hierbei soll vor allem überprüft werden, ob das in Kapitel 5.2.2 erläuterte full text indexing mit Lucene Performance-Verbesserungen mit sich bringt. Dazu wird eine Query auf verschieden großen Dateien getestet.

Die Ausführung der Benchmarks erfolgte auf einem BenQ Joybook mit 1,4 GHz Pentium mobile Prozessor und 512 MB DDR RAM.
Die Jamrox-Middleware wurde auf der Postgres-Datenbank mit dem JDBC-Treiber „postgresql-8.1-404“ ausgeführt, wobei die Datenbank mit der postmaster-Option -F gestartet wurde. Hier wurde bereits in der Version nach [Sie05] festgestellt, dass diese Einstellung den Parsing-Vorgang beschleunigt.
Außerdem wurde, wie auch schon in der Version nach [Sie05], in der Datenbank auf der `parent`-Spalte ein Index angelegt. Das Anlegen des Indexes wurde wie in Kapitel 5.3.1 beschrieben verändert.
eXist wurde in der neuesten Version 1.1 betrieben.
Im Folgenden sollen die Ergebnisse der Benchmarks grafisch dargestellt und analysiert werden.
Abbildung 17 zeigt die Ergebnisse des Testablaufs A1. Dieser zeigte, dass das Einlesen der XML-Dokumente mit der Version nach [Sie05] schneller ging als mit der neuen Version der Middleware.
Das liegt daran, dass die neue Version für jeden Textknoten ein eigenes Datenbank-Tupel angelegt wird. In der Version nach [Sie05] wurde der Text noch als Wert eines Elementknotens gespeichert. Diese Änderung führt dazu, dass doppelt so viele Tupel in der Datenbank abgelegt werden als vorher.



	1 MB	3 MB	10 MB
Jamrox Version KP	1,777	32,627	91,812
Jamrox neu mit TS	3,335	59,384	188,101
Jamrox neu ohne TS	3,572	67,321	204,057
eXist	2,472	12,718	OutOfMemoryError

Abbildung 17: A1

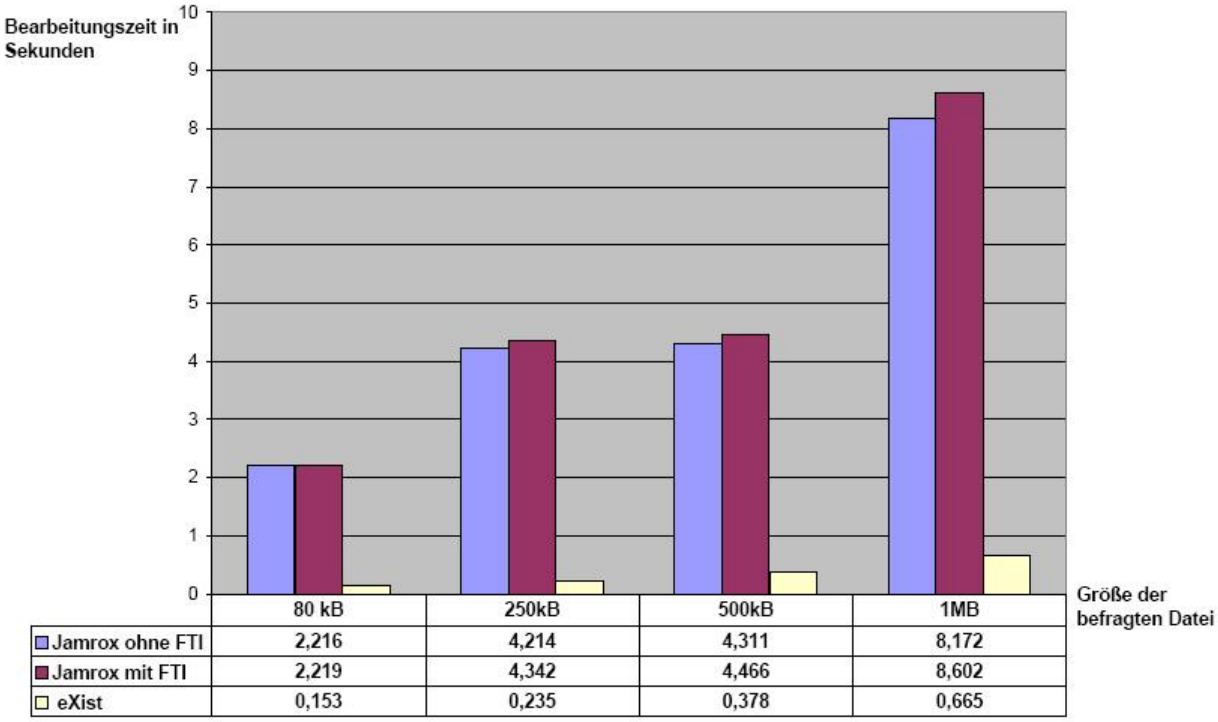
Die Einführung des TransactionServices führte zu Performance-Verbesserungen, welche allerdings erst bei größeren Dateien wirklich erkennbar werden. Beim Einlesen einer 3MB-Datei war die Middleware mit dem TransactionService 8 Sekunden schneller als ohne, beim Einlesen einer 10MB-Datei waren es knapp 16 Sekunden.
Das Einlesen der Dateien mit eXist erfolgte bei der 1MB-Datei etwa um ein Drittel schneller, bei der 3MB-Datei etwa fünfmal schneller als das Einlesen mit Jamrox in einer Transaktion. Allerdings konnte die 10MB-Datei mit eXist nicht eingelesen werden, weil ein „OutOfMemory“-Error ausgelöst wurde.
Um die Performance des XUpdateQueryServices zu bewerten, wurde jeweils ein Benchmark für die Jamrox-Middleware und für eXist entworfen, in dem eine XUpdate-Datei eingelesen und das Update auf einer Resource ausgeführt wird.
In diesem Update kommen folgende Update-Konstrukte vor:

- `xupdate:append`
- `xupdate:rename`
- `xupdate:insert-after`
- `xupdate:insert-before`
- `xupdate:update`

Die Ausführung des Updates dauerte mit Jamrox ohne TransactionService 1,082 Sekunden, mit TransactionService 1,073 Sekunden und mit eXist 0,1843 Sekunden. Das Update wurde mit eXist folglich fast sechsmal schneller durchgeführt als mit Jamrox.
Mit dem letzten Testablauf A3 sollte die Performanceverbesserung beim Ausführen von Queries mit „contains“ durch Hinzufügen von full text indexing überprüft werden. Außerdem sollte ein weiterer Vergleich mit eXist durchgeführt werden.
Andere Queries wurden nicht mit eXist verglichen, da dies mit der Version nach [Sie05] bereits getan wurde und es in der neuen Version keine grundlegenden Veränderungen im XPathQueryService gab.
Die XPath-Anfrage, die für den Testablauf benutzt wurde, sieht folgendermaßen aus:

```
//child:node()[contains(text(),'excepted')]
```

Die Anfrage wurde an Dateien verschiedener Größe gestellt, um festzustellen, ob sich der Effekt des full text indexing auf die Performance bei größeren Dateien verstärkt.
Die Ergebnisse der Messungen sind in Abbildung 18 dargestellt.



	80 kB	250kB	500kB	1MB
Jamrox ohne FTI	2,216	4,214	4,311	8,172
Jamrox mit FTI	2,219	4,342	4,466	8,602
eXist	0,153	0,235	0,378	0,665

Abbildung 18: A3

Es stellte sich heraus, dass sich durch das full text indexing mit Lucene keine Performanceverbesserungen ergaben. Die Query wurde ohne FTI sogar etwas schneller ausgeführt als mit FTI. Auch bei größeren Dateien (die vor allem mehr und längere Textknoten enthalten), hatte das FTI keinen positiven Effekt auf die Performance.
Dass keine großen Performance-Verbesserungen zu erwarten sind, wurde bereits in Kapitel 5.2.2 vorhergesagt, da die Arbeitsweise und Navigation von Jaxen nicht grundlegend verändert wurde.
Es muss jeder Textknoten betrachtet werden: ohne FTI wird jeder Textknoten mit „indexOf“ nach dem Suchstring durchsucht, mit FTI wird für jeden Textknoten die Id erfragt und diese mit den Id's der Treffer aus dem Index verglichen.
Aus den Ergebnissen der Benchmarks kann gefolgert werden, dass der einmalige Zugriff auf den Index und der Vergleich der Id's länger dauert als das Durchsuchen des Strings mit `indexOf`.
Aufgrund dieser Ergebnisse wurde der für das FTI hinzugefügte Code aus der Middleware auskommentiert. Er bleibt so für Testzwecke oder zur Weiterentwicklung erhalten.
Die Abarbeitung der Query wurde in eXist wieder sehr viel schneller erledigt als mit der Middleware. Beim Befragen der 250kB-Datei war eXist fast 18 Mal schneller als Jamrox.
Im Folgenden soll die Funktionsweise der eXist-Query Engine etwas näher betrachtet werden. Dabei soll deutlich gemacht werden, wo die Stärken von eXist gegenüber Jamrox liegen.
eXist stellt eine eigene Implementierung einer Query Engine bereit, die von einem effizienten Index-Schema unterstützt wird. Die Indizierung ermöglicht eine schnelle Identifizierung von strukturellen Beziehungen zwischen Knoten.
Um die Beziehung zwischen einem Paar von gegebenen Knoten festzustellen, müssen die Knoten nicht in den Speicher geladen werden.
Dies bringt den Vorteil, dass die „teure“ Operation, einen Knoten von einem persistenten Speicher zu laden, nicht nötig ist [eXi06a]. eXist verwendet für die Speicherung eine Abwandlung von DOM, das sich PDOM (persistent DOM) nennt [eXi06c].
Um die Indizierung zu ermöglichen, wird jeder Knoten mit einer eindeutigen Id versehen. Die Id wird so vergeben, dass es möglich ist, die Beziehung zwischen zwei Knoten im gleichen Dokument allein von der Id herzuleiten. Dies wird in der neuesten Version von eXist mit hierarchischen Id's umgesetzt.
Das heißt, dass der Wurzelknoten die Id 1 hat und allen Kindknoten die Id's 1.1, 1.2, 1.3 usw. zugeordnet werden. Bei der Indizierung wird jeder qualifizierte Name eines Elements oder Attributs auf eine Liste von Id's abgebildet, deren zugehörige Knoten diesen Namen besitzen.
Wenn die Id's zu einem Element mit Namen A und die Id's zu einem Element mit Namen B bekannt sind, kann ein XPath-Ausdruck „A/B“ mit einer einfachen Join-Operation zwischen der Knotenmenge A und B berechnet werden [eXi06a]. Der Zugriff auf den DOM-Baum ist dafür nicht nötig.
Des Weiteren benutzt eXist einen „fulltext index“ und einen „range index“. Der „fulltext index“ bildet Text-Tokens auf die Text- und die Attribut-Knoten ab, in denen sie vorkommen. Der Index wird automatisch erstellt und gewar-

tet.
Der „range index“ basiert auf dem Datentyp von bestimmten Knotenwerten und ermöglicht einen schnelleren Zugriff auf Knoten über ihren Wert. Dies funktioniert analog zu Indizes, die auf relationalen Datenbanken erstellt werden. Diese Indizes können vom Benutzer erzeugt und konfiguriert werden [eXi06a].

7 Ausblick

In Kapitel 1 wurden die verschiedenen Möglichkeiten für die Weiterentwicklung einer Software genannt. Nun sollen zum Abschluss dieser Arbeit die zur Weiterentwicklung der Middleware Jamrox durchgeführten Maßnahmen zusammengefasst werden.
Zur Verbesserung der Qualität (perfektive Wartung) wurden folgende Änderungen und Erweiterungen an der Middleware durchgeführt:

- Das EdgeInline-Schema wurde so verändert, so dass nun alle XML-Konstrukte in der Datenbank gespeichert werden und somit kein Verlust von Informationen durch die Abbildung auf die Datenbank entsteht.
  Es werden folgende XML-Bestandteile nun ebenfalls unterstützt: Namensräume, CDATA-Abschnitte, Kommentare, Processing-Instructions und Entity-Referenzen. Außerdem werden in der neuen Version alle Text-Kindknoten eines Elements in der Datenbank abgelegt.

- Der Inhalt einer Resource kann nun mit SAX oder DOM gesetzt werden bzw. als SAX oder DOM zugänglich gemacht werden.

- Die Middleware wurde so erweitert, dass nun ein TransactionService und ein XUpdateQueryService zur Verfügung stehen.

- Ein DB-Client, der eine grafische Oberfläche bietet, wurde an die Datenbank angebunden.

- Die Performance der Middleware beim Importieren von Dokumenten konnte durch den Einsatz des TransactionService verbessert werden.

- Die Testfälle zur Sicherung der Funktionaliät der Middleware wurden überarbeitet.

- Die Formatierung des Quellcodes wurde vereinheitlicht.

- Der Quellcode wurde so angepasst, dass er nun in einer „bereinigten“ Form vorliegt. Das heißt, dass der Code anerkannten Coding-Standards entspricht und keine Bugs enthält.

- Die Dokumentation des Quellcodes wurde ergänzt und vereinheitlicht.

Bei der Weiterentwicklung wurden Fehler entdeckt, so dass die Middleware auch korrektiv gewartet wurde. Die Implementierung der Middleware entspricht nach der Fehlerbeseitigung nun vollständig der Spezifikation der XML:DB-API.
Bezüglich der perfektiven Wartung konnten folgende Punkte leider nicht umgesetzt werden:

- Das Einbinden einer inkrementellen Validierung eines Dokuments vor einem Update scheiterte daran, dass die Implementierung der dafür vorgesehenen Bibliotheken nur mit der kommerziellen DOM-Implementierung der Firma Oracle benutzt werden kann.

- Die Performance der Datenbank während der Bearbeitung von XPath-Anfragen konnte nicht gesteigert werden. Hier zeigte sich, dass die Performance stark von der Arbeitsweise der Jaxen XPath Engine abhängig ist, welche im Rahmen dieser Arbeit nicht grundlegend verändert werden sollte.

Weitere Ideen zur Weiterentwicklung der Middleware sind:

- Bisher wird nur XPath 1.0 von der DB unterstützt. Hier wäre eine Erweiterung auf XPath 2.0/XQuery [W3C06e] denkbar. Dies ist allerdings mit Jaxen bisher nicht möglich, da Jaxen nur XPath 1.0 untersützt.
  eXist implementierte hierfür beispielsweise einen eigenen Service (XQueryService) [exi06d].

- Der in der neuen Version der Middleware hinzugefügte XUpdateQueryService wurde mit XUpdate implementiert. Eine Anpassung an den W3C-Standard XQueryUpdate [W3C06f] wäre in weiterführenden Arbeiten denkbar.
  eXist bietet bereits eine Erweiterung des XQueryServices für XQueryUpdate [exi06d].

Eine Übersicht anderer Middlewares, die XML-Dokumente in relationalen Datenbanken speichern, ist auf der Website von Ronald Bourret [Bou06] aufgeführt. Es gibt hier allerdings keine mit Jamrox vergleichbare Middleware, da die meisten der aufgeführten Produkte auf einer Schemadefinition (DTD oder einem XML Schema) basierend die XML-Dokumente in Objekte einer Zielsprache übersetzen.
Es gibt außerdem keine Middleware, die wie Jamrox die Schnittstelle der XML:DB-API implementiert. Auf der Website der XML:DB Initiative befindet sich zwar ein Hinweis auf eine Java-Implementierung für JDBC-Datenquellen [XML06a], die von Jeremias Maerki entwickelt wurde. Der dazugehörige Link funktioniert allerdings nicht und auf der Website von Jeremias Maerki konnte kein Verweis auf eine derartige Implementierung gefunden werden.
Abschließend kann die Middleware folgendermaßen bewertet werden: Die Eigenschaft der Jamrox-Middleware, alle möglichen Formate von XML-Dokumenten speichern zu können, bringt eine große Flexibilität. Das Parsen der Dokumente mit SAX bringt den Vorteil, dass beliebig große Dokumente eingelesen werden können.
Die Jaxen XPath Engine konnte gut in die Middleware integriert werden und bringt zusätzlich den nützlichen Effekt, dass die XML-Daten als DOM zur Verfügung stehen.
Durch die XML:DB-API-Schnittstelle kann die Middleware problemlos an XML:DB-API-konforme Clients angebunden werden.
Außerdem ist die Middleware unabhängig von der darunter liegenden Datenbank, da mit JDBC eine Schnittstelle zur Datenbank gewählt wurde, deren Treiber je nach Datenbank angepasst werden kann.
Andererseits birgt die Flexibilität beim Format der XML-Dokumente den Nachteil, dass schon beim Importieren von kleinen XML-Dokumenten eine lange Datenbanktabelle entsteht. Dies führt dazu, dass beim Bearbeiten von XPath-Anfragen aufwändige Datenbankzugriffe nötig sind.

Weitere Performance-Einbuße bei den XPath-Queries haben ihre Ursache in der Arbeitsweise der XPath Engine Jaxen. Diese ist nicht optimal auf die Speicherung der Daten ausgerichtet und muss so für jedes durch eine XPath-Achse ausgewählte Tupel auf die Datenbank zugreifen und den zugehörigen DOM-Knoten erzeugen. Diese „teure“ Operation wird beispielsweise in eXist (siehe Kapitel 6) durch die Indizierung von Knoten vermieden.
Das Hauptziel, das mit dem Entwurf der Middleware verfolgt wurde, war es **beliebige** XML-Dokumente in einem relationalen Schema zu speichern. Dieses Ziel wurde im Rahmen dieser Arbeit erreicht, es gehen keine Informationen mehr verloren. Darüberhinaus konnte die Middleware hinsichtlich weiterer Aspekte wie Funktionalität, Performance und Zuverlässigkeit verbessert und erweitert werden.

Literatur

[Apa06a] Apache Lucene Project. Lucene, 1.10.2006. `http://lucene.apache.org/`.

[Apa06b] Apache Lucene Project. Lucene FAQ, 1.10.2006. `http://wiki.apache.org/jakarta-lucene/LuceneFAQ`.

[Bea06] Bea Systems. XML, 1.10.2006. `http://dev2dev.bea.com/xml/stax.html`.

[Bec06] Beck, Kent; Gamma, Erich. JUnit, Testing Resource for Extreme Programming, 1.10.2006. `http://www.junit.org`.

[Bou06] Bourret, Ronald. XML Database Products, 1.10.2006. `http://www.rpbourret.com/xml/XMLDatabaseProds.htm`.

[Bur06] Burn, Oliver. Checkstyle 4.2, 1.10.2006. `http://checkstyle.sourceforge.net/`.

[Cod06a] Codehaus. Jaxen FAQ, 1.10.2006. `http://jaxen.org/faq.html`.

[Cod06b] Codehaus. Jaxen Project, 1.10.2006. `http://jaxen.org/`.

[eXi06a] eXist-db.org. Configuring Database Index, 1.10.2006. `http://exist-db.org/indexing.html`.

[eXi06b] eXist-db.org. eXist - Open Source Native XML Database, 1.10.2006. `http://www.exist-db.org`.

[eXi06c] eXist-db.org. Package org.exist.dom, 1.10.2006. `http://exist.sourceforge.net/api/org/exist/dom/package-summary.html`.

[exi06d] exist-db.org. XQuery, 1.10.2006. `http://exist-db.org/xquery.html`.

[Gam95] Gamma, Erich; Helm, Richard, Johnson; Ralph, E. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, Boston, 1995.

[Gen04] Genevès, Pierre. Improving Efficiency of XPath-Based XML Querying, 2004. `http://wam.inrialpes.fr/publications/2004/toward-xpath-efficiency.pdf`.

[Gos05] Gospodnetic, Otis; Hatcher, Erik. *Lucene in action*. Manning Publications Co., Greenwich, 2005.

[Har03] Harold, Elliotte Rusty. *Processing XML with Java*. Addison-Wesley, Boston, 2003.

[Har04] Harold, Elliotte Rusty; Means, W. Scott . *XML in a Nutshell*. O'Reilly, Beijing, 2004.

[Jet06] Jetbrains. IntelliJ 5.1, 1.10.2006. `http://www.jetbrains.com/idea/`.

[Kos99] Kossmann, Donald; Florescu, Daniela. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database, 1999. `http://wwwbayer.in.tum.de/cgi-webcon/webcon/lehrstuhldb/details/Veroeffentlichungen/num/30/1`.

[Nie04] Niedermeier, Stephan. *Cocoon 2 und Tomcat*. Galileo Press, Bonn, 2004.

[Nov06] Novalis. GNU Lesser General Public License, 1.10.2006. `http://www.gnu.org/licenses/lgpl.html`.

[Ora06] Oracle. Package org.w3c.dom.validation, 1.10.2006. `http://www.oracle.com/technology/docs/tech/xml/xdk/doc_library/beta/doc/java/javadoc/org/w3c/dom/validation/package-summary.html`.

[Pro04] Projektpraktikum. *XML-Datenbanken: Entwicklung einer Middleware*. Universität Koblenz-Landau, Koblenz, 2004.

[See03] Seemann, Michael. *Native XML Datenbanken im Praxiseinsatz*. Software & Support Verlag GmbH, Frankfurt, 2003.

[Sie05] Sieren, Klaus-Peter. *XML-Datenbanken: Effiziente XPath-Anfragen auf relational gespeicherten XML-Dokumenten*. Universität Koblenz-Landau, Koblenz, 2005.

[Sou06a] Sourceforge.net. Events vs. Trees, 1.10.2006. `http://www.saxproject.org/event.html`.

[Sou06b] Sourceforge.net. FindBugs - Find Bugs in Java Programs, 1.10.2006. `http://findbugs.sourceforge.net/`.

[Sou06c] Sourceforge.net. SAX 2.0 Extensions, 1.10.2006. `http://www.saxproject.org/sax2-ext.html`.

[Sun06a] Sun. Hierarchy For Package org.w3c.dom, 1.10.2006. `http://java.sun.com/j2se/1.5.0/docs/api/org/w3c/dom/package-tree.html`.

[Sun06b] Sun Microsystems Inc. Code Conventions for the Java Programming Language, 1.10.2006. `http://java.sun.com/docs/codeconv/`.

[Sun06c] Sun Microsystems Inc. Code Conventions for the Java Programming Language, 1.10.2006. `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`.

[Sun06d] Sun Microsystems Inc. Java SE - Java Database Connectivity (JDBC), 1.10.2006. `http://72.5.124.55/javase/technologies/database.jsp`.

[Swa90] Swanson, E. Burton ; Beath, Cynthia Mathis . Departmentalization in software development and maintenance. *Commun. ACM*, 33(6):658–667, 1990.

[The04] Theis, Fabian; Hardt, Manfred. *Suchmaschinen entwickeln mit Apache Lucene*. Software & Support Verlag GmbH, Frankfurt, 2004.

[Tit06] Titanium Group. Titanium, 1.6.2006. `http://titanium.dstc.edu.au/xml/xmldbgui/`.

[Voo06] Voohra, Deepak. Using DOM 3.0 Validation Techniques with XDK 10g, 1.10.2006. `http://www.devx.com/Java/Article/31221`.

[W3C06a] W3C. Document Object Model (DOM) Level 3 Validation Specification, 1.10.2006. `http://www.w3.org/TR/2002/WD-DOM-Level-3-Val-20021008/DOM3-Val.html`.

[W3C06b] W3C. DOM Level 3 Validation implementation report, 1.10.2006. `http://www.w3.org/2003/07/DOM-Level-3-Val-implementations.html`.

[W3C06c] W3C. XML Path Language, 1.10.2006. `http://www.w3.org/TR/xpath`.

[W3C06d] W3C. XML Path Language - deutsche Uebersetzung, 1.10.2006. `http://www.informatik.hu-berlin.de/~obecker/obqo/w3c-trans/xpath-de/`.

[W3C06e] W3C. XQuery 1.0 and XPath 2.0 Functions and Operators, 1.10.2006. `http://www.w3.org/TR/xpath-functions/`.

[W3C06f] W3C. XQuery Update Facility Requirements, 1.10.2006. `http://www.w3.org/TR/xqupdate/`.

[XML06a] XML:DB Initiative. Application Programming Interface (API) for XML Databases, 1.10.2006. `http://xmldb-org.sourceforge.net/xapi`.

[XML06b] XML:DB Initiative. Class Database Manager, 1.10.2006. `http://xmldb-org.sourceforge.net/xapi/api/index.html`.

[XML06c] XML:DB Initiative. TransactionService, 1.10.2006. `http://xmldb-org.sourceforge.net/xapi/api/org/xmldb/api/modules/TransactionService.html`.

[XML06d] XML:DB Initiative. What's new, 1.10.2006. `http://xmldb-org.sourceforge.net/news.html`.

[XML06e] XML:DB Initiative. XMLResource, 1.10.2006. `http://xmldb-org.sourceforge.net/xapi/api/org/xmldb/api/modules/XMLResource.html`.

[XML06f] XML:DB Initiative. XPathQueryService, 1.10.2006. `http://xmldb-org.sourceforge.net/xapi/api/org/xmldb/api/modules/XPathQueryService.html`.

[XML06g] XML:DB Initiative. XUpdate, 1.10.2006. `http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html`.

[XML06h] XML:DB Initiative. XUpdateQueryService, 1.10.2006. `http://xmldb-org.sourceforge.net/xapi/api/org/xmldb/api/modules/XUpdateQueryService.html`.

Abbildungsverzeichnis