Diplomarbeit

Anwendungsstudie TransForm

Implementierung eines transaktionssicheren Datenbankmodells für das Internet

Nils Andre

Eingereichte Diplomarbeit gemäß den Bestimmungen der Prüfungsordnung der Universität Freiburg für den Diplomstudiengang Informatik vom 29.11.2003

Institut für Informatik
Lehrstuhl für Datenbanken und Informationssysteme
Albert-Ludwigs-Universität Freiburg
Freiburg im Breisgau

Autor Nils Andre

Bearbeitungszeit 26. April 2007 bis 26. Oktober 2007

Gutachter Prof. Dr. Georg Lausen

Betreuer Dipl.-Inf. Matthias Ihle
Dipl.-Inf. Martin Weber
Lehrstuhl für Datenbanken und Informationssysteme

Erklärung nach §10(7) der Diplomprüfungsordung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Arbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, den 18. Oktober 2007

Nils Andre

Zusammenfassung

Diese Arbeit präsentiert die schrittweise Neuimplementierung einer Software zur Analyse von Unternehmensrisiken im Kontext eines XML-Datenzugriffsmodells und TransForm. Das Datenmodell erlaubt es, relationale Datenbanken wie XML-Dokumente anzusprechen und zu manipulieren. TransForm ist ein neues Zugriffsmodell für Benutzereingaben in Web-Formularen, das direkt in XHTML-Code eingebettet werden kann. Mittels einer an XPath angelehnten Sprache werden dort Eingabeelemente in Formularen direkt an Datenbankobjekte gebunden. Diese Architektur ermöglicht es, dass Transaktionen die gesamte Sitzung einer Webseite umspannen, und mehrere Benutzer voneinander isoliert Datenmanipulationen durchführen können.

In dieser Arbeit wird die Entwicklung des XML-Datenmodells vorgestellt und eine konkrete Implementierung als TransForm-Anwendung dokumentiert und spezifiziert. TransForm erlaubt es, nahezu jede Web-Anwendung mit Transaktionsmanagement, das den ACID-Eigenschaften genügt, zu versehen. Ziel der Arbeit ist es, zu untersuchen, ob diese Modelle für kommerzielle Software – in diesem Fall die Unternehmensrisikoanalyse – einsetzbar sind.

Stichwörter: Web-Anwendungen, Formulare, XML-Datenbanken, XPath, Transaktionen, Parallelitätskontrolle

Abstract

This thesis presents the stepwise re-implementation of a piece of software that can handle corporate risks' analysis within the context of an XML data access model and TransForm. This data access model specifies how relational databases can be addressed and manipulated as if they were XML documents. TransForm is a novel user interface for user inputs in web formulae, which can directly be embedded into XHTML code. With an XPath-alike language, input elements in formulae can be bound to database objects. This architecture allows transactions to span the whole session of a website and multiple users to manipulate data being isolated from each other.

In this work, we present the deployment of the XML data model and document the concrete implementation as a TransForm application and it's specification. TransForm allows nearly every web application being endowed with transaction management that guarantees the ACID properties. The goal of this thesis is to investigate on how these models can be applied to commercial software, in this case the risks' analysis as mentioned above.

Keywords : web applications, forms, XML databases, XPath, transactions, concurrency control

Danksagung

Inhaltsverzeichnis

Kapitel 1.

Einleitung

Die Verarbeitung und Bearbeitung von Daten hat – einhergehend mit der Verbreitung des Internets und sukzessiv ansteigenden Datenübertragungsmengen – einen hohen Stellenwert eingenommen: Daten müssen effizient gespeichert, übertragen und bearbeitet werden können. In den allermeisten Anwendungen übernimmt die Speicherung der Daten eine relationale Datenbank, während der Transfer von Daten zwischen Anwendungen und verschiedenen Datenbanksystemen mittels *Extensible Markup Language*, kurz XML, bewerkstelligt wird. XML ist ein in den letzten Jahren zu einem Standardaustauschformat gewordener Ansatz, Daten mittels einer Metasprache strukturiert und logisch zusammenhängend zu repräsentieren.

Die Bearbeitung von Daten, insbesondere für Anwender ohne Programmierkenntnisse, stellt jedoch sehr hohe Anforderungen an zusätzliche Software, und steht konträr zu dem relationalen Ansatz, in dem lediglich Relationen und deren Tupel ohne semantische Zusammenhänge bearbeitet werden können. Sollen semantische Zusammenhänge in die Bearbeitung von Daten mit einfließen, so wird in der Regel ein *Content-Management-System* (kurz „CMS“), ein Verwaltungssystem für Inhalte, benötigt. Ein solches System stellt eine „Zwischenschicht“ zwischen der Datenbank auf der einen und dem Benutzer auf der anderen Seite dar, um sicherzustellen, dass der Benutzer sich einerseits nicht um formale Dinge wie korrekte Fremd- und Primärschlüssel kümmern muss, und andererseits die Daten, welche Zusammenhänge zwischen Relationen und Tupeln darstellen, vom System intern verwaltet und für den Benutzer möglichst einfach aufbereitet werden können.

Daten werden im Internet durch den Austausch von Informationen mittels Aktionen zwischen Client und Server bearbeitet. Typischerweise werden während einer serverseitigen Aktion Daten aus einer Datenbank ausgelesen und in Formularen auf einer Webseite auf dem Client dargestellt. Diese können dann durch den Benutzer verändert und mittels einer weiteren Serveraktion in die Datenbank zurückgeschrieben werden. Das Resultat dieser Aktion wird abermals durch eine Webseite auf dem Client angezeigt. Dieser „Zyklus“ umfasst in der Regel mehrere unzusammenhängenden Transaktionen auf Serverseite. Wünschenswert wäre jedoch, diese Transaktionen

in einer zusammenhängenden Transaktion, welche die gesamte Sitzung umspannt, zusammenzufassen. Das in Kapitel 3 vorgestellte Zugriffsmodell mit TransForm stellt eine Möglichkeit dar, diese Anforderung ohne die Notwendigkeit eines zusätzlichen Transaktionsservers zu realisieren.

In der vorliegenden Arbeit soll anhand eines konkreten Szenarios ein System implementiert werden, welches Daten einer XML-Sicht bearbeiten kann. Hierzu soll untersucht werden, wie relationale Datenbanken auf XML-Dokumente abgebildet werden können und inwiefern TransForm als Schnittstelle zwischen Clients und allgemeinen Datenbankschemata einen möglichst einfachen Datenzugriff erlaubt.

1.1. Anwendungsszenario

Folgendes Szenario soll in der Diplomarbeit als Leitbeispiel dienen, um die Entwicklung und Anwendung der vorgestellten Datenstrukturen und deren Manipulationsmöglichkeiten zu skizzieren: Es wird angenommen, man wolle eine Unternehmens-Risikoanalyse durchführen; hierzu sollen in einer Datenbank Mandanten (= Firmen oder Teilunternehmen) verwaltet werden, die man anhand ausgewählter Bereiche bewerten und analysieren kann. Es sollen Bereiche definiert werden können, welche sich wiederum in einzelne Potentiale untergliedern; diese Potentialbereiche sollen spezifisch gewichtet und separat bewertet werden können. Des Weiteren sollen solche Analysen in jedem Geschäftsjahr vorgenommen werden können, um die Ergebnisse der einzelnen Analysen vergleichen zu können. Der letzte Punkt soll dazu dienen, Informationen nicht nur bearbeiten, sondern diese auch visualisieren zu können.

Das Anwendungsszenario ist entnommen aus einer proprietären Software, der CORSuite, ein Produkt der COR GmbH, Freiburg, und soll als illustrierende Anwendung dienen, um zu untersuchen, ob sich die in dieser Arbeit vorgestellten Daten- und Datenzugriffsmodelle auf kommerzielle Software anwenden lassen.

1.2. Problemformulierung und Ziel der Diplomarbeit

In der Diplomarbeit werden verschiedene Fragestellungen aufgeworfen und untersucht, welche im Zusammenhang mit dem Wechselspiel von XML und relationalen Datenbanken sowie deren Zugriffsmodellen und einer konkreten Software-Implementierung stehen. Hier sind unter Anderem folgende zu nennen:

- Da TransForm mit XPath-Ausdrücken arbeitet, muss untersucht werden, wie sich aus einer relationalen Datenbank ein XML-Dokument ableiten lässt, so dass möglichst viele semantische Zusammenhänge der Daten in XML abgebil-

det werden. Inwiefern können Schlüssel- und Fremdschlüsselbeziehungen für einen solchen Prozess herangezogen werden?

- Welche Vorteile und Probleme ergeben sich daraus, dass Datenbanken wie XML-Dokumente adressiert werden können? Inwiefern erlaubt ein Zugriffsmodell über eine XML-Zwischenschicht, Daten in einer darunterliegenden Datenbank zu manipulieren?

- Wie können derartige Manipulationen in Formularen auf Webseiten dargestellt und verarbeitet werden? Kann ein Prozess des Bearbeitens über die gesamte Sitzung auf einer Webseite in einer Datenbank-Transaktion zusammengefasst werden?

- Erlauben die sich aus den vorherigen Fragestellungen ergebenden Lösungsansätze die Umsetzung in ein praktisches Szenario?

Das in dieser Diplomarbeit entwickelte System soll den Anforderungen sowohl an intuitiver Bearbeitbarkeit als auch an ACID-Eigenschaften von Datenbanktransaktionen genügen. Hierzu wird das im vorherigen Absatz formulierte Szenario in ein ER-Diagramm, in ein relationales Schema, und dann in Software entwickelt. Das Szenario wird – wie ein roter Faden – immer wieder aufgegriffen, um die konkreten Probleme und Fragestellungen näher zu erörtern; die entwickelte Software wird jedoch so allgemein wie möglich gehalten und konfigurierbar sein, um in anderen, ähnlichen Szenarien Einsatz finden zu können.

1.3. Aufbau der Diplomarbeit

Die Diplomarbeit gliedert sich in fünf Kapitel (Abb. 1.1).

Nach der Einleitung werden in Kapitel 2 zunächst einige Grundlagen zu Datenbanken und XML behandelt und deren Zusammenhänge aufgezeigt. Des Weiteren wird das bereits in der Einleitung skizzierte Beispielszenario aufgegriffen und konkret entwickelt (mittels ER-Diagramm und relationaler Implementierung). Ferner wird eine Transformationsmöglichkeit von relationalen Datenbanken in XML-Dokumente vorgestellt und auf das Beispielszenario angewendet, sowie ein XPath-ähnliches Datenzugriffsmodell für virtuelle XML-Bäume, denen eine relationale Datenbank zugrunde liegt, entwickelt.

Anschließend wird in Kapitel 3 TransForm vorgestellt, das ein neuartiges Modell der Benutzerzugriffe auf Formulare darstellt. Zuerst wird TransForm in den Kontext von Formulareingaben auf Webseiten eingebettet, sowie die Implementierung der Komponenten sowohl auf Client- als auch auf Serverseite erörtert und die sich

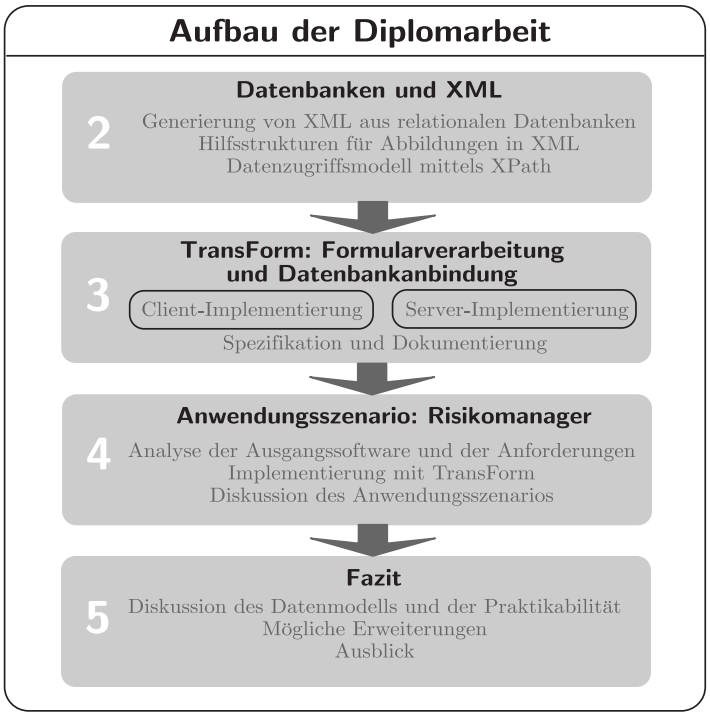


Abb. 1.1.: Aufbau der Diplomarbeit

ergebenden Vorteile sowie Probleme diskutiert. In den letzten Abschnitten dieses Kapitels werden die von TransForm unterstützten spezifischen Browser-Tags beschrieben und die Unterschiede zur Original-Spezifikation herausgestellt.

In Kapitel 4 wird die detaillierte Entwicklung des Anwendungsszenarios in Software behandelt. Zunächst werden die Anforderungen an die Software analysiert, dann wird die konkrete Implementierung vorgestellt, und beschrieben, wie das Datenzugriffsmodell mittels XPath sowie TransForm in das Szenario eingebettet und konfiguriert wird. Abschließend werden die verwendeten Designschablonen und Erweiterungen beschrieben, sowie ein kurzes Fazit zur Anwendung hinsichtlich Erweiterbarkeit und Benutzbarkeit gezogen.

Kapitel 5 beinhaltet ein Fazit, eine abschließende Diskussion und einen Überblick über mögliche Erweiterungen.

Im Anhang finden sich technische Aspekte sowie ein Überblick über die Struktur der Implementierung und die verwendeten Teilkomponenten.

1.4. Konventionen

Einige Textbausteine dieser Arbeit sind vom normalen Schriftbild abgehoben, um spezielle Namen, Ausdrücke, Quelltexte etc. hervorzuheben. Die Gestaltung dieser wurde an die Konventionen des O'Reilly Verlags [17] angelehnt.

- *Kursiv*: Dieser Schrifttyp wird für wörtliche Zitate aus Büchern und Artikeln, Definitionen, Ersterwähnungen von Begriffen, wichtige Teilsysteme und Komponenten, wichtige Software-Systeme, Attribute von Relationen sowie für hervorgehobenen Text genutzt.
- **Fettschrift**: Im Fließtext bezeichnet ein in Fettschrift gesetzter Text einen Relationsbezeichner, d.h. den Namen einer Relation; am Anfang eines Absatzes oder einer Aufzählung eine wichtige Komponente oder ein prägnantes Wort.
- ⟨*Spitzgeklammerte Ausdrücke*⟩: Diese werden im jeweiligen Kontext für entweder eine Ersetzung (beispielsweise eines Parameters), die der Benutzer vornehmen muss, oder für einen generellen Platzhalter verwendet.
- `Nichtproportionalschrift`: Dieser Schrifttyp wird für Quelltexte, prägnante Dateinamen, Attribute in XML-Dokumenten, sowie für Funktionen oder Ausdrücke von JavaScript- oder PHP-Programmen genutzt.
- `Nichtproportionalschrift mit linkem Rand` wird für Rückgaben von AJAX-Requests benutzt.
- Kapitälchen: Dieser Schrifttyp wird für Grammatikregeln und prägnante Aktionen des TransForm-Servers genutzt.
- In Aufzählungen dienen die Symbole ✓ und ✗ der Indizierung einer tendenziell positiven bzw. negativen Bewertung.

Um Missverständnisse zu vermeiden, wird ferner die folgende begriffliche Konventionen benötigt:

- Im Folgenden bezeichnet der Begriff „XPath-Ausdruck“ einen Ausdruck der Form, wie dieser in Kapitel 3 eingeführt wird. Da sich diese Form von Ausdrücken semantisch erheblich von konventionellen XPath-Ausdrücken [25] unterscheidet, jedoch einen zentralen Aspekt der vorliegenden Diplomarbeit darstellt, wird dieser im Folgenden schlicht „XPath-Ausdruck“ genannt; sollte ein XPath-Ausdruck im Sinne von [25] gemeint sein, so wird dieser als „echter XPath-Ausdruck“ oder „XPath 1.0-Ausdruck“ gekennzeichnet.

Kapitel 2.

Datenbanken und XML

Dieses Kapitel widmet sich der Beziehung von relationalen Datenbanken und dem XML-Modell. Hierzu werden zunächst knapp einige Grundlagen und Begriffe eingeführt, auf die in der vorliegenden Arbeit oft Bezug genommen wird. Des Weiteren wird ein Algorithmus zur Transformation von relationalen Datenbankschemata in XML-Dokumente vorgestellt. Ziel dieses Algorithmus' ist es, eine virtuelle XML-Struktur über einer Menge von Relationen (und ihrer Instanzen) zu erzeugen, um dann auf dieser Struktur eine an XPath 1.0 angelehnte Sprache zu entwerfen, die es erlaubt, Datenbanken wie XML-Dokumente behandeln zu können. Ferner wird das in der Einleitung vorgestellte Szenario detailliert entwickelt und an das neue Datenmodell angepasst.

2.1. Relationale Datenbanken

Das relationale Datenbankmodell wurde von Edgar F. Codd 1970 [2] erstmals vorgeschlagen und ist heute, trotz einiger Kritikpunkte, ein etablierter Standard zum Speichern von Daten. Zur mathematischen Beschreibung dieses Modells dient üblicherweise die *Relationenalgebra* [2, 9, 11], die hier um einige Aspekte erweitert wird.

Definition 1 (**Datenbankschema, Relationsschema**). *Ein Datenbankschema $\mathcal{S}$ ist definiert als eine Menge von Relationsschemata.*

$$\mathcal{S} = \{R_1, \ldots, R_n\}$$

Jedes Relationsschema R ist definiert über einer Menge von Attributen, genauer geschrieben als

$$R(A) = R(\{A_1, \ldots, A_m\})$$

Für jedes Relationsschema $R(A)$ sei

- *$r \subseteq tup(A)$ eine beliebige Instanz zu R, d.h. eine Menge von Tupeln über dem Wertebereich der Attribute der Relation,*

- $P_R \subseteq A$ *der Primärschlüssel der Relation* R,
- $A_R = \{A_1, ..., A_n\}$ *eine Kurzschreibweise für die Attribute der Relation* R,
- $F_R \subseteq \mathcal{P}(A)$ *die Menge der Fremdschlüssel der Relation* R,
- $\phi : F_R \to \mathcal{S}$ *die Funktion, welche für einen gegebenen Fremdschlüssel* f *die referenzierte Tabelle liefert, und*
- $\rho : F_R \to \mathcal{P}(A)$ *die Funktion, welche für einen gegebenen Fremdschlüssel* f *die Attribute der referenzierten Tabelle liefert.*[1]

Die zwei letzten Definitionen über Fremdschlüssel sind hier vorgestellte Erweiterungen der Relationenalgebra, da anhand der Fremdschlüsselbeziehungen zwischen den Relationen später ein dem Schema entsprechendes XML-Dokument generiert werden soll. Ferner werden die Operatoren für die relationale Projektion mit π, die Selektion mit σ, die Umbenennung von Attributen mit δ, der Verbund (Join) mit $\bowtie$ und das kartesische Produkt mit $\times$ bezeichnet und in ihrer in [11] definierten Bedeutung verwendet.

2.2. XML

XML (Abkürzung für „*Extensible Markup Language*“, [24]) ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdateien.

Solche Textdateien, genannt *XML-Dokumente*, bestehen aus (öffnenden und schließenden) *Tags* der Form

<tag attr$_1$="wert$_1$" ... attr$_n$="wert$_n$"> ⟨*Inhalt*⟩ </tag>

Tags bezeichnet man in ihrer Gesamtheit als *Elemente*, die mit *Attributen* attr$_i$ ausgezeichnet werden können. ⟨*Inhalt*⟩ kann aus weiteren (geschachtelten) Tags (Elementen) oder beliebigen Zeichenketten bestehen[2]. Ebenfalls existiert die Kurzschreibweise <tag ... /> für Tags, deren Inhalt leer ist. Des Weiteren muss in jedem XML-Dokument genau ein Tag existieren, welches in keinem anderen Tag enthalten ist; es wird als *Wurzelelement* bezeichnet.

Eine weitere wichtige Eigenschaft von XML-Dokumenten ist die *Wohlgeformtheit*. Ein XML-Dokument heißt *wohlgeformt*, wenn es sämtliche XML-Regeln einhält, also

[1] Das Symbol $\mathcal{P}(A)$ steht hier für die Potenzmenge von A.

[2] Elemente mit gemischtem Inhalt, d.h. sowohl Zeichenketten als auch Tags, werden hier ausgeschlossen.

die Tags in der oben genannten Weise formatiert sind, es ein Wurzelelement enthält, sowie alle Start- und End-Tags mit Inhalt ebenentreu-paarig verschachtelt sind.[3]

Ursprünglich wurde XML entwickelt, um den Herausforderungen zu begegnen, die sich aus elektronischem Publizieren im großen Stil ergaben [3]. Mittlerweile spielt es eine immer größere Rolle beim Datenaustausch, sowohl im Web, als auch zwischen Anwendungen, da es ein internationalisiertes, medienunabhängiges elektronisches Veröffentlichen von Daten ermöglicht, die semantisch zusammenhängend in einem XML-Dokument zusammengefasst werden können. Solche Dokumente sind sowohl für Computer einfach zu verarbeiten, als auch für Menschen (relativ) einfach zu lesen. Ferner wird mittels XSLT[4] [26] eine mächtige Möglichkeit zur Verfügung gestellt, XML-Dokumente umzuformatieren, und so beispielsweise für eine Ausgabe im Browser in HTML darzustellen. Um XML-Dokumente formal behandeln zu können, werden noch einige Definitionen benötigt:

Definition 2 (**XML-Baum**[5])**.** *Seien EL eine Menge von Elementtypen, Att eine Menge von Attributen, gekennzeichnet durch ein führendes '@'-Zeichen, Σ^* die Menge aller Zeichenketten und V eine Menge von Objekt-Identifikatoren. Ein XML-Baum ist gegeben zu $T = (V, E, lab, ele, att, root)$, wobei*

- *V eine endliche Menge von Knoten, $root \in V$,*
- *$E = \{(v_1, v_2) \mid v_1 \in V, v_2 \in ele(v_1)\}$ definiert die Elter-Kind-Relation der Kanten des Baumes,*
- *$ele : V \to \mathcal{P}(V) \cup \Sigma^*$ eine partielle Funktion, welche zu einem Knoten v die Menge seiner Nachfolgeknoten oder (im Falle eines Textknotens) seinen Wert zurückliefert,*
- *$att : V \times Att \to \Sigma^*$ eine partielle Funktion, welche zu einem gegebenen Knoten v und einem Attributbezeichner a dessen Wert zurückliefert, und*
- *$lab : V \to EL$ die Funktion, welche zu einem Knoten v den Elementtyp zurückliefert.*

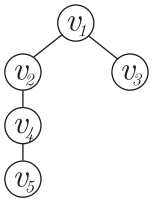
Beispiel 1 (XML-Baum)**.** Dem XML-Dokument aus Abb. 2.1 wird ein Baum zugeordnet.

[3] Im Folgenden werden alle XML-Dokumente als wohlgeformt vorausgesetzt.

[4] *Extensible Stylesheet Language Transformation*

[5] Diese Definition ist angelehnt an die der Vorlesungsfolien zur Vorlesung Datenbanken 2, gelesen von Prof. Dr. Georg Lausen im Sommersemester 2007, `http://download.informatik.uni-freiburg.de/lectures/DB2/2007SS/Slides/05_XMLNormalform.pdf` , Seite 10.

```
<main>
  <knoten id="1">
     <subknoten text="Irgendwas">
     Inhalt
     </subknoten>
  </knoten>
  <knoten id="2" />
</main>
```



$root = v_1$, $lab(v_1) = main$
$lab(v_2) = lab(v_3) = Knoten$
$lab(v_4) = subknoten$, $lab(v_5) = \#text$
$att(v_2, @id) =$ ‘1‘, $att(v_3, @id) =$ ‘2‘
$att(v_4, @text) =$ ‘$Irgendwas$‘
$ele(v_1) = \{v_2, v_3\}$, $ele(v_2) = \{v_4\}$
$ele(v_5) =$ ‘$Inhalt$‘

Abb. 2.1.: XML-Dokument und XML-Baum zu Beispiel 1

2.3. Generierung eines XML-Dokumentes aus einem Datenbankschema

In diesem Abschnitt soll analysiert werden, wie aus einem gegebenen Datenbankschema (d.h. einer Menge von Relationsschemata) sowie der gegebenen Beziehungen zwischen den Relationen ein XML-Baum generiert werden kann.

Im Vergleich zur reinen generischen Transformation von Relationen bzw. deren Instanzen in XML, bei der lediglich für jede Instanz jedes Tupel in XML „übersetzt“ wird (beschrieben in [11, 18, 10]), soll ein XML-Abbild die Zusammenhänge der Daten möglichst „gut“ in XML repräsentieren. Objektbeziehungen sollten auch nach einer XML-Transformation bestehen bleiben, Objekte also über *Enthaltensein-Beziehungen* ineinander geschachtelt werden.[6]

Zunächst sollen jedoch noch weitere Prämissen an ein solches XML-Abbild formuliert werden:

- Das XML-Abbild soll in der Praxis lediglich eine „virtuelle“ Transformation sein; d.h. über dem relationalen Datenbankmodell wird eine virtuelle Schicht implementiert, welche den Datenzugriff erlaubt, „als ob man auf ein XML-Dokument zugreife“.

- Redundanzen im Abbild sind vorerst unerheblich, da nicht das Ziel erreicht werden soll, das generierte XML-Dokument tatsächlich zu materialisieren. Zunächst soll entworfen werden, wie ein solches Abbild tatsächlich aussehen kann.

- Der Zugriff auf das Abbild soll mittels XPath, bzw. einem XPath-ähnlichen Derivat möglich sein, d.h. mit ggf. einer Teilmenge der echten XPath 1.0-Funktionalität auskommen. Diese Sprache soll leicht in eine Datenbankanfragesprache wie SQL übersetzt werden können.

[6] Ist ein Objekt o logisch einem anderen Objekt p untergeordnet, und es nicht sinnvoll ist, o bearbeiten zu wollen, ohne p zu spezifizieren, so sollte dies durch eine Enthaltensein-Beziehung der Form `<p><o/></p>` ausgedrückt werden.

Der hier vorgestellte, an [8] angelehnte Ansatz verwendet die Schlüssel- und Fremdschlüsselbeziehungen zwischen Relationsschemata, um eine möglichst „gute“ Abbildungen in XML zu erhalten (in dieser Arbeit wird allerdings auf eine Abbildung in DTDs[7] oder XML-Schemata verzichtet). Primärschlüssel einer Relation erlauben keine Rückschlüsse über Beziehungszusammenhänge zwischen Relationen, da ein Primärschlüssel sich immer nur auf genau ein Relationsschema bezieht. Fremdschlüssel stellen jedoch Beziehungen zwischen Tabellen dar, welche im Folgenden genauer untersucht werden sollen. Hierzu seien zwei Relationsschemata

$$R(\{A_1, \ldots, A_i\}) \text{ und } S(\{B_1, \ldots, B_j\})$$

gegeben, sowie P_R der Primärschlüssel zu R und $f \in F_S$ ein Fremdschlüssel in S mit $\phi(f) = R$. R bezeichnet hier die Elter-Relation, S die Kindrelation. Ferner seien r und s jeweils beliebige Instanzen zu R bzw. S.

Die Fremdschlüsselbedingung impliziert, dass $\pi[f]s \subseteq \pi[P_R]r$ gilt. Von Interesse ist daher die Kardinalität der Beziehungen, d.h. wie viele Tupel aus s auf Tupel aus r abgebildet (referenziert) werden. Hier sind im Wesentlichen drei Fälle möglich:

1. Zu jedem Tupel aus r existiert höchstens bzw. genau ein Tupel aus s (schwache bzw. strikte $1:1$-Beziehung)

2. Es existieren $n \geq 0$ Tupel aus s, welche ein Tupel aus r referenzieren ($1:n$-Beziehung)

3. Mittels einer Hilfsrelation lässt sich der Sachverhalt ausdrücken, dass $n \geq 0$ Tupel aus s gerade $m \geq 0$ Tupel aus r referenzieren ($m:n$-Beziehung)

Da jedoch das Datenbankschema unabhängig von dessen Instanzen analysiert werden soll, kann man in diesem Fall nicht die Kardinalität der Mengen bestimmen. Um die Beziehungskardinalitäten zu erfahren, müssen die Fremdschlüsselbeziehungen genauer betrachtet werden. Hier sind folgende Fälle möglich:

1. Ist der Fremdschlüssel $f_S = P_S = P_R$, also gleichzeitig sowohl Fremd- als auch Primärschlüssel in S und Primärschlüssel in R, muss eine $1:1$-Beziehung zwischen R und S vorliegen.

2. Ist f_S Fremdschlüssel, so liegt eine $1:n$-Beziehung vor.

3. Liegt ein Relationsschema $H(A_1 \cup A_2)$ vor, wobei $A_1 = F_R$ Fremdschlüssel aus R und $A_2 = F_S$ Fremdschlüssel aus S ist, so realisiert diese Hilfsrelation gerade eine $m:n$-Beziehung zwischen zwischen R und S.

[7] *Document Type Definition*-Dateien sind Dateien, die Strukturinformationen zu einer XML-Datei aufnehmen.

Ausgehend von diesen Beobachtungen kann nun das *XML-Abbild* induktiv definiert werden. Sei hierzu im Folgenden $\mathcal{S}$ beliebig, und $R(A), S(B), H(C) \in \mathcal{S}$, sowie r, s, h Instanzen zu den jeweiligen Relationsschemata.[8] Das XML-Dokument wird nun wie folgt gebildet:

1. Das Wurzelelement ist `<schema>`.

2. Falls für ein R gilt, dass $F_R = \emptyset$, so bilde für jedes Tupel $t \in r$ ein Kindelement zu `<schema>` der Form:

   `<r` $\langle a_1 \rangle$`="`$\langle \pi[a_1]t \rangle$`"` ... $\langle a_1 \rangle$`="`$\langle \pi[a_k]t \rangle$`" />`

   wobei alle Attribute von `<r>` gerade die Attribut-Werte-Paare des Tupels t darstellen ($a_i \in A, 1 \leq i \leq k$)

3. Falls für ein R gerade $F_R = a = \{a_1, \ldots, a_n\}$ gilt, und $\phi(a) = S$ (also genau ein Fremdschlüssel in R existiert), so verfahre wie in Schritt (2), verwende jedoch als Elterelement für die neu zu bildenden Elemente für jedes Tupel t gerade diejenigen `<s>`-Elemente, für die gilt $\pi[a]t = \pi[\rho(a)]s$. Hierbei kann insbesondere $R = S$ gelten, in diesem Fall werden diejenigen Elemente direkte Kindelemente von `<s>`, welche der Bedingung $\pi[a]t = \{NULL\}$ genügen.

4. Falls $F_R = \{a_1, \ldots, a_n\}$ für ein R ist und $\phi(a_i) \neq \phi(a_j), 1 \leq i < j \leq n$, also die Fremdschlüssel a_i unterschiedliche Relationen referenzieren, ist genauer zu untersuchen, welche Elemente die Elterelemente aller Tupel $t \in r$ werden. Dies kann wie folgt geschehen:

   - Ist $n = 2$, $\phi(a_1) = R_1$ und $\phi(a_2) = R_2$, so handelt es sich um eine Hilfsrelation im Sinne von Fall (3) aus der letzten Aufzählung. In diesem Fall müssen die Tupel von R_2, die nach Regel (3) gebildet werden, gerade Kinder von R_1 werden, und umgekehrt.
   - Ist $n > 2$, werden also mehr als zwei Relationen durch die Relation R verknüpft, so kann man entweder den „passenden“ Fremdschlüssel auswählen, der die Elter-Kind-Beziehung am besten beschreibt, oder für jeden Fremdschlüssel das Elterelement wählen, welches durch den Fremdschlüssel beschrieben wird (letzteres Verfahren sorgt für viel Redundanz, die aber im Falle eines „virtuellen“ XML-Abbildes nicht störend ist)

Dieses Vorgehen hat den Vorteil, dass bei einer $1 : n$- bzw. $n : m$-Beziehungskomplexität zwischen zwei Relationen der Abbildungsprozess sehr einfach ist. Stehen jedoch mehr als zwei Relationen in einer Beziehung zueinander, so muss die Zuordnung gegebenenfalls „von Hand“ erfolgen.

[8] Zirkuläre Fremdschlüsselbedingungen seien hier ausgeschlossen.

2.4. Anwendung auf das Szenario

Um diese Verfahrensweise zu illustrieren, greifen wir das Beispiel aus der Einleitung auf: Man möchte mittels einer Datenbank Firmen und deren Abteilungen hinsichtlich gewisser gewichteter Kriterien bewerten und eine Unternehmensanalyse durchführen. Hierzu werden mehrere Relationen benötigt:

- Jede zu analysierende Firma oder Firmeneinheit wird durch einen *Mandanten* repräsentiert; jeder Mandant besteht für dieses Beispiel lediglich aus einem Namen.
- Eine Analyse soll für jeden Mandanten über mehrere Jahre durchgeführt werden, um Analysen miteinander vergleichen und Entwicklungen aufzeigen zu können.
- Jede (jährliche) Analyse soll in Einzelbereiche unterteilt sein, die wiederum in Potentiale aufgeteilt sind. Die Potentiale eines Bereiches sollen unterschiedlich gewichtet werden können.
- Durch eine Bewertung sollen alle Potentiale mit einem Wert zwischen 0 und 100 „versehen“ werden können.

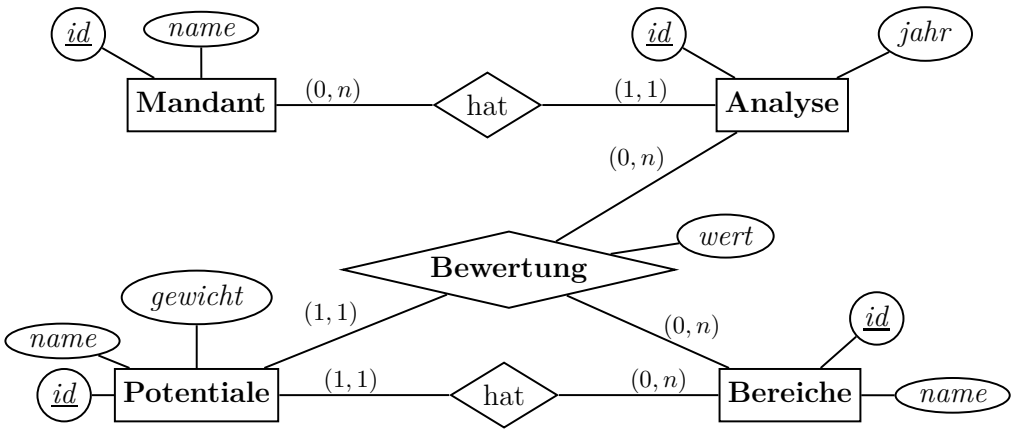


Abb. 2.2.: Eine erster Entwurf als ER-Diagramm

Ein erster Entwurf als *Entity-Relationship-Diagramm* (kurz *ER-Diagramm* [1]) sowie eine mögliche Instantiierung des Datenbankschemas sind in Abbildung 2.2 bzw. Abbildung 2.3 aufgeführt. In dieser Instanz existieren zwei Analysebereiche, „Vertrieb“ und „Buchhaltung“. Die Potentiale von Vertrieb sind „Lieferzeit“ und „Effizienz“, welche zu jeweils 30 bzw. 70 Prozent in eine Bewertung eingehen. In der Analyse von 2007 des Mandanten „Maier“ werden diese Potentiale mit jeweils 70 bzw. 100 Punkten bewertet.

Mandant

id	*name*
1	Maier
2	Bonn

Analyse

id	⊣*m_id*[9]	*jahr*
1	1	2006
2	1	2007

Bereiche

id	*name*
1	Vertrieb
2	Buchhaltung

Potentiale

id	⊣*b_id*	*name*	*gewicht*
1	1	Lieferzeit	30%
2	1	Effizienz	70%
3	2	Vollständigkeit	50%
4	2	Leistung	50%

Bewertung

⊣*a_id*	⊣*b_id*	⊣*p_id*	*wert*
2	1	1	70
2	1	2	100
2	2	3	80
2	2	4	70

Abb. 2.3.: Eine mögliche Instantiierung zum ER-Diagramm aus 2.2

Relation	F (Fremdschlüssel)	P (Primärschlüssel)
Mandant	$\emptyset$	$\{id\}$
Analyse	$\{m\_id\}$ $\phi(\{m\_id\}) = Mandant$, $\rho(\{m\_id\}) = Mandant.id$	$\{id\}$
Bereiche	$\emptyset$	$\{id\}$
Potentiale	$\{b\_id\}$ $\phi(\{b\_id\}) = Bereiche$, $\rho(\{b\_id\}) = Bereiche.id$	$\{id\}$
Bewertung	$\{a\_id, b\_id, p\_id\}$ $\phi(\{a\_id\}) = Analyse$, $\rho(\{a\_id\}) = Analyse.id$ $\phi(\{b\_id\}) = Bereiche$, $\rho(\{b\_id\}) = Bereiche.id$ $\phi(\{p\_id\}) = Potentiale$, $\rho(\{p\_id\}) = Potentiale.id$	$\{a\_id, b\_id, p\_id\}$

Abb. 2.4.: Fremd- und Primärschlüssel zum ER-Diagramm aus Abb. 2.2

Wendet man die induktive Definition des XML-Dokumentes aus dem vorherigen Abschnitt auf das Beispiel an, so ergibt sich das XML-Abbild wie folgt: Für die Relation **Mandant** werden demnach alle Tupel direkte Kindknoten von `<schema>`, alle Tupel der Relation **Analyse** werden wiederum Kindknoten der dazugehörigen `<mandant>`-Knoten. Für die Relation **Bereiche** existiert kein Fremdschlüssel, d.h. alle Tupel dieser Relation werden ebenfalls Kindknoten von `<schema>`. Alle Tupel der Relation **Potentiale** werden durch die eindeutige Fremdschlüsselbeziehung Kindknoten der jeweiligen `<bereiche>`-Knoten.

Problematisch erweist sich in diesem Beispiel das Ergebnis für die Relation **Bewertung**. Dort sind drei Fremdschlüssel zu verzeichnen, welche unterschiedliche Relationen referenzieren. Daher kann hier nach der vorgeschlagenen Methode in Punkt

[9]Hierbei soll das Symbol „⊣“ indizieren, dass es sich bei dem ausgezeichneten Attribut um einen Fremdschlüssel handelt.

(4) entweder der jedes Tupel dieser Relation den jeweilig referenzierten Knoten untergeordnet werden, oder aber manuell gewählt werden, welcher Fremdschlüssel die Beziehung „am Besten“ repräsentiert. Letzterer Methode wurde im folgenden XML-Abbild nachgegeben:

```
<schema>
    <mandant id="1" name="Maier">
        <analyse id="1" jahr="2006" />
        <analyse id="2" jahr="2007">
            <bewertung bid="1" pid="1" wert="70" />
            <bewertung bid="1" pid="2" wert="100" />
            <bewertung bid="2" pid="3" wert="80" />
            <bewertung bid="2" pid="4" wert="70" />
        </analyse>
    </mandant>
    <bereiche id="1" name="Vertrieb">
        <potentiale id="1" name="Lieferzeit" gewicht="30" />
        <potentiale id="2" name="Effizienz" gewicht="70"  />
    </bereiche>
    <bereiche id="2" name="Buchhaltung">
        <potentiale id="3" name"Vollständigkeit" gewicht="50" />
        <potentiale id="4" name"Leistung" gewicht="50" />
    </bereiche>
</schema>
```

Abb. 2.5.: XML-Abbild zu dem Beispiel aus Abschnitt 2.4

2.5. Virtuelle XML-Abbilder mittels Hilfsstrukturen

Im vorherigen Abschnitt wurde gezeigt, dass das vorgeschlagene Verfahren im allgemeinen Fall sehr viel Redundanz im XML-Abbild erzeugen kann. Um diese zu vermeiden, muss der Benutzer eingreifen, und in solchen Fällen eine entsprechende Beziehung „auswählen“. Ferner wird durch einen solchen Algorithmus lediglich ein statisches Abbild erzeugt. Dieses müsste immer dann neu generiert werden, wenn sich Daten in der Datenbank verändern.

Es ist jedoch wünschenswert, die Repräsentation eines relationalen Schemas in XML zu „fixieren“, ohne hier eine Fixierung auf die Daten der Datenbank zu erreichen. Das XML-Abbild soll lediglich als „virtuelle Schicht“ über das relationale Modell gelegt werden. Hierzu wird daher eine Hilfsstruktur benötigt, welche dauerhaft die Abbildung des relationalen Schemas in XML definiert. Diesen Zweck erfüllt beispielsweise ein zusätzliches XML-Dokument, welches die hierzu notwendigen In-

formationen aufnimmt. Diese Vorgehensweise besitzt Vor- und Nachteile gleichermaßen:

- ✓ Eine solche Hilfsstruktur ist leicht konfigurierbar und ermöglicht große Flexibilität bei der Modellierung eines XML-Abbildes.
- ✓ Sind die relationalen Zusammenhänge relativ einfach, d.h. lassen sich die Enthaltensein-Beziehungen ohne Zutun des Benutzers herausfinden, so können diese Informationen als Basis für eine Hilfsstruktur verwendet werden.
- ✓ Zusätzliche Meta-Informationen über die relationale Struktur, etwa solche, die die Bearbeitbarkeit der Daten aus Sicht eines Benutzers erleichtern, können dort integriert werden.
- ✗ Es wird zusätzlich Platz für Informationen verwendet und einige Informationen werden redundant gespeichert; ändert sich etwas an der Struktur des zu Grunde gelegten Datenbankschemas, so müssen innerhalb des Strukturdefinitionsdokuments zusätzliche Anpassungen vorgenommen werden, um weiterhin ein korrektes XML-Abbild zu garantieren. Daher ergibt sich ein Mehraufwand für die Verwaltung der Datenbank.

Das Konzept der Hilfsstruktur erweist sich als praktisch, wenn viele zusätzliche Meta-Informationen zu einer Datenbank vorliegen, und man die in der Datenbank gespeicherten Daten möglichst intuitiv für den Benutzer aufbereiten und präsentieren möchte, so dass die Handhabung der Bearbeitung von Daten hierdurch erleichtert wird. [10] Die Ziele dieser Hilfsstruktur sind somit:

1. Sie fasst alle benötigten Informationen zusammen, um ein Datenbankschema wie ein XML-Dokument behandeln zu können.
2. Das Abbild bleibt „virtuell“, so dass aufgrund einer Hilfsstruktur die Daten lediglich andersartig adressiert werden können. Anfragen werden direkt von einer an XPath 1.0 angelehnten Sprache in SQL übersetzt.

Eine solche Hilfsstruktur wird mittels dem in Abb. 2.6 auf Seite 17 dargestellten Algorithmus XmlStructure gebildet. Dieser erhält als Eingabe ein Datenbankschema $\mathcal{S}$ wie in Definition 1, und gibt ein Hilfsdokument in Form eines XML-Baumes H aus. Der Algorithmus fügt für jede Relation R des Datenbankschemas $\mathcal{S}$, die keine Fremdschlüssel besitzt, einen Knoten unterhalb des Wurzelknotens ein (erste Schleife), und setzt dessen Attribute und Primärschlüsselinformationen. Zuerst müssen alle Relationen ohne Fremdschlüssel dem XML-Baum hinzugefügt werden,

[10] Kapitel 4 gibt hierfür ein ausführliches Beispiel.

Algorithmus 1. XmlStructure($\mathcal{S}$)

Eingabe: Datenbankschema $\mathcal{S} = \{R_1, \ldots, R_n\}$ wie in Definition 1
Ausgabe: Hilfsdokument $H = (V, E, lab, ele, att, root)$

1: $EL \leftarrow \{schema, R_1, \ldots, R_n\}$
2: $Att \leftarrow \{@primary\_key, @join\_condition, @rel\}$
3: $V \leftarrow \{root\}$; $lab(root) \leftarrow schema$
4: **for all** $R \in \mathcal{S}$ **with** $F_R = \emptyset$ **do**
5: // *R besitzt keine Fremdschlüssel*
6: $v \leftarrow newnode()$
7: $V \leftarrow V \cup \{v\}$ // *Knotenmenge aktualisieren*
8: $ele(root) \leftarrow ele(root) \cup \{v_i\}$ // *Kind von root*
9: $lab(v) \leftarrow R$ // *Elementtyp setzen*
10: $att(v, `rel`) \leftarrow$ `R` // *Attribute setzen*
11: $att(v, `primary\_key`) \leftarrow P_R$
12: **end for**
13: **for all** $R \in \mathcal{S}$ **with** $F_R \neq \emptyset$ **do**
14: // *R besitzt Fremdschlüssel, diese werden nun betrachtet*
15: **for all** $f = \{b_1, \ldots, b_k\} \in F_R$ **do**
16: $S \leftarrow \phi(f)$
17: $A \leftarrow \rho(f) = \{a_1, \ldots, a_k\}$
18: **for all** $v' \in V$ **do**
19: **if** $lab(v') = S$ **then**
20: $v \leftarrow newnode()$
21: $V \leftarrow V \cup \{v\}$
22: $ele(v') \leftarrow ele(v') \cup \{v\}$ // *Elter-Kind-Beziehung*
23: $lab(v) \leftarrow R$ // *Elementtyp setzen*
24: $att(v, `rel`) \leftarrow$ `R` // *Attribute setzen*
25: $att(v, `primary\_key`) \leftarrow P_R$
26: $att(v, `join\_condition`) \leftarrow$ `%parent.a`$_1$`=%this.b`$_1$, ...,`%parent.a`$_k$`=%this.b`$_k$
27: **end if**
28: **end for**
29: **end for**
30: **end for**
31: **return** H

Abb. 2.6.: Algorithmus XmlStructure

da sich Relationen mit Fremdschlüsseln möglicherweise auf diese beziehen. Danach wird jede Relation betrachtet, die Fremdschlüssel besitzt (zweite Schleife). Für jeden Fremdschlüssel wird jeweils ein Kindknoten unterhalb der referenzierenden Relation gebildet und die Fremdschlüsselinformationen in Zeile 26 hinzugefügt.

Hieraus folgt, dass der Wurzelknoten des XML-Hilfsdokuments `<schema>` ist und für jede Relation $R \in \mathcal{S}$ wenigstens ein Knoten `<r />` existiert. Wenn R einen Fremdschlüssel in R' besitzt, ist der jeweiligen Elterknoten der Knoten, dem die Relation R' zugeordnet wurde, und andernfalls ist der Elterknoten `<schema>`. So wird nur die Struktur der Daten, nicht aber die Daten selbst, gespeichert. Nach dem Durchlauf des Algorithmus XmlStructure können die durch ihn erzeugten Redundanzen im Abbild entfernt werden. Um eine eindeutige Darstellung von Adressierungsmöglichkeiten in der Struktur zu erhalten, sollte für jede Relation genau ein Knoten vorhanden sein, d.h. eventuelle Redundanzen müssen auch hier „von Hand“ entfernt werden. Für dieses Beispiel kann eine Hilfsstruktur wie folgt aussehen:

```
1 <schema>
2     <mandant rel="mandant" primary_key="id" title="Mandanten">
3         <analyse rel="analyse" primary_key="id"
4             join_condition="%this.mid␣=␣%parent.id" >
5             <bewertung rel="bewertung" primary_key="a_id,b_id,p_id"
6                 join_condition="%this.bid␣=␣%parent.id" />
7         </analyse>
8     </mandant>
9     <bereiche rel="bereiche" primary_key="id"
10        title="Analysebereiche">
11        <potentiale rel="potentiale" primary_key="id"
12            join_condition="%this.bid␣=␣%parent.id" />
13    </bereiche>
14 </schema>
```

Abb. 2.7.: XML-Hilfsdokument zu dem Beispiel aus Abschnitt 2.4

Dieses Dokument, nachfolgend *Strukturdokument* oder *Hilfsdokument* genannt und namentlich als `structure.xml` gespeichert, kann einmal durch den Algorithmus XmlStructure berechnet und dann manuell angepasst werden. Einmal korrekt entworfen, lassen sich aus diesem Dokument alle Informationen extrahieren, die benötigt werden, um XPath-Ausdrücke korrekt in SQL zu übersetzen und somit einen Zugriff auf die Datenbank mittels XPath-Ausdrücken zu erlauben. Auf dieses Datenzugriffsmodell soll im folgenden Abschnitt näher eingegangen werden.

2.6. Datenzugriff mittels XPath

XPath (*XML Path Language*, [25]) ist eine vom W3C[11] festgelegte und standardisierte Sprache, um Teile eines XML-Dokumentes zu *lokalisieren*, d.h. diese zu adressieren, oder auf einem XML-Dokument basierende Werte wie Strings, Integer oder Wahrheitswerte zu berechnen.

XPath basiert auf der Baumrepräsentation eines XML-Dokuments und ermöglicht das Navigieren durch diesen Baum durch das *Lokalisieren* von Knoten mittels zahlreicher Kriterien. XPath wurde ursprünglich entwickelt, um eine (gemeinsame) Sprache für die Kommunikation innerhalb und zwischen XPointer und XSLT einzuführen. Jedoch etablierte sich XPath aufgrund seiner kompakten und konsequenten Notation schnell in vielen weiteren Anwendungsgebieten [18].

In dieser Arbeit soll eine eigene, an XPath 1.0 angelehnte Sprache entworfen werden, deren Ausdrücke sich leicht in SQL transformieren lassen, und die dabei möglichst einfach in ihrer Syntax und Struktur bleibt. Im Folgenden wird immer der Begriff „XPath“ im Sinne der neu eingeführten Notation verstanden; andernfalls wird von einem „gewöhnlichen XPath-Ausdruck“ gesprochen.

Definition 3 (**XPath-Ausdrücke**). *Ein XPath-Ausdruck ist eine Zeichenkette beginnend mit dem Zeichen* `/`*, gefolgt von einer durch das Zeichen* `/` *getrennten Folge von „Lokalisationsschritten“, bestehend aus Knotentest K_i und optionalen Prädikaten p_i. Formal ist ein XPath-Ausdruck also gegeben als*

$$/K_0p_0/ \ \ldots \ /K_np_n/K_{n+1}$$

Ein solcher XPath-Ausdruck ist immer absolut (im Sinne von XPath 1.0) und der Knotentest K_i bezieht sich zunächst auf die XML-Hilfsstruktur, selektiert also innerhalb dieses XML-Baumes den Knoten k_i.[12] Die durch die Auswertung eines solchen Ausdrucks gewonnenen Informationen werden später mit Hilfe der gegebenen Prädikate p_i dafür verwendet, aus dem XPath-Ausdruck einen SQL-Ausdruck zu gewinnen, welcher auf die der Hilfsstruktur zugrunde liegende Datenbank angewendet werden kann.

Infolgedessen selektiert ein solcher XPath-Ausdruck gerade eine Menge von Tupeln einer Relation, genau ein Tupel oder einen atomaren Wert.

Definition 4 (**Knotentests**). *Ein Knotentest K_i ist eine Zeichenkette, deren Inhalt der Name jedes Knotens des XML-Hilfsdokumentes sein kann. Im Falle von $i = n+1$*

[11] World Wide Web Consortium. Internationale Organisation, die Standards und Regeln für das Internet definiert

[12] Im Gegensatz zu XPath 1.0-Ausdrücken gemäß W3C-Spezifikation werden hier keine anderen Achsen als die `child::`-Achse unterstützt.

(„letzter“ Knotentest eines Ausdrucks) kann K_{n+1} eine Kommata-separierte Liste von Attributknoten enthalten, d.h. $K_{n+1} =$ `@a`$_1$`,...,@a`$_j$. Diese bezieht sich dann auf den Knoten k_n und selektiert genau die aufgeführten Attribute der diesem Knoten zugrunde liegenden Relation. Ist K_{n+1} leer, so werden alle Attribute der Relation des Knotens k_n selektiert.

Definition 5 (**Prädikate**). *Prädikate p_i sind durch die Zeichen* `[` *und* `]` *eingefasste Zeichenketten (Strings), welche durch die folgende Grammatik definiert werden:*

$$\textsc{Pred} \longrightarrow \texttt{(}\textsc{Pred}\ \texttt{\&\&}\ \textsc{Pred}\texttt{)} \mid \texttt{(}\textsc{Pred}\ \texttt{||}\ \textsc{Pred}\texttt{)} \mid \texttt{!(}\textsc{Pred}\texttt{)} \mid \textsc{Atom}$$

wobei

- $\textsc{Atom} \rightarrow \texttt{(}\textsc{Expr}\ \theta\ \textsc{Expr}\texttt{)}$ *(Relationsoperator)*
- $\theta \rightarrow$ `==` | `!=` | `<=` | `>=` | `<` | `>` *(Relationen)*
- $\textsc{Expr} \rightarrow f\texttt{(}\textsc{Expr}_1\texttt{, ..., }\textsc{Expr}_n\texttt{)} \mid \textsc{Var} \mid \textsc{Const}$ *(Funktionen, $n \geq 0$)*
- $\textsc{Var} \rightarrow \langle$*Variablenbezeichner*$\rangle$ *(Variablen)*
- $\textsc{Const} \rightarrow \langle$*Integer*$\rangle$ | `"`$\langle$*String*$\rangle$`"` | `'`$\langle$*String*$\rangle$`'` *(Konstanten)*

Beispiel 2 (Prädikate)**.** Einige gültige Prädikate sind beispielsweise:

- $p_1 =$ `(id == 3 && name == 'schmidt')`
- $p_2 =$ `(f(1, g(h(x))) < 1)`
- $p_3 =$ `(position() > 0 && position() < 100)`

Die Operatoren `&&` bzw. `||` stehen für *Konjunktion* bzw. *Disjunktion* zweier Ausdrücke, und `!` für den Negationsoperator. Sie werden im Folgenden – um Prädikate mathematisch behandeln zu können – mit $\wedge$ bzw. $\vee$ und $\neg$ bezeichnet. Ferner sei für einen Prädikatsausdruck p gerade $Var(p)$ die Menge aller Variablenbezeichner, die in ihm vorkommen. Die Semantik der XPath-Ausdrücke ist nun wie folgt: Sei *root* der Wurzelknoten des XML-Baumes des Hilfsdokuments H und $ele(k)$ für einen gegebenen Knoten k die Menge dessen Kindknoten. Ein XPath-Ausdruck $x =$`/`K_0p_0`/` $\ldots$ `/`K_np_n`/`K_{n+1} ist nun für ein gegebendes Hilfsdokument H gültig, $H \models x$, wenn:

- $K_0 = \{k_0\} = \{root\}, lab(root) =$ `'schema'` und $p_0 =$ `''`.
- $K_i = \{k_i\} \subseteq ele(k_{i-1})$, $lab(k_i) \in \mathcal{S}$ und $Var(p_i) \subseteq A_{lab(k_i)}$, $1 \leq i \leq n$.
- Falls $K_{n+1} =$`@a`$_1$`,...,@a`$_j$, so muß $p_{n+1} =$`''` und $\{a_1, \ldots, a_j\} \subseteq A_{lab(k_n)}$ sein

Prädikate beschränken innerhalb eines Ausdrucks die Menge der selektierten Tupel, während durch den Knotentest jeweils die zu selektierenden Objekte bzw. deren Beziehungen zueinander ausgedrückt werden. Prädikate werden verwendet, um in SQL die `WHERE`-Bedingung eines `SELECT`-, `UPDATE`- oder `DELETE`-Statements zu formulieren.

In Prädikatsausdrücken können iterierte Konjunktionen sowie iterierte Disjunktionen kürzer geschrieben werden, indem überflüssige Klammern in Ausdrücken der Form `(`a_1 `&&` ... `(`a_{n-1} `&&` a_n`)...)` weggelassen werden, also als `(`a_1 `&&` ... `&&` a_n`)` geschrieben werden.

Ferner verhalten sich die Operatoren links-assoziativ, es gibt keinen Vorrang von gewissen Operatoren über anderen (üblicherweise bindet ein logisches „und“ stärker als ein logisches „oder“ – dies ist hier jedoch *nicht* der Fall). Ein Ausdruck der Form `a || b && c` verhält sich also als ob dieser korrekt geklammert als `(a || b) && c` geschrieben würde.

Beispiel 3 (XPath-Ausdrücke)**.** Ein gültiger XPath-Ausdruck, bezogen auf die in Abb. 2.7 dargestellte Hilfsstruktur ist beispielsweise

```
/schema/mandant[id==1]/analyse[id==2]/bewertung[pid==1 && bid==2]/@wert
```

Dieser Ausdruck bezeichnet gerade das Attribut `wert` des Tupels der Relation **Bewertung**, das dem Prädikat `pid == 1 && bid == 2` genügt, und deren jeweilige „Vorfahren“ in den Relationen **Mandant** und **Analyse** den Prädikaten `id == 1` bzw. `id == 2` genügen.

Definition 6. *Eine (atomare) Datenbankoperation, also eine Operation auf der Datenbank mit darübergelegter XML-Schicht, ist definiert als ein Tripel* (a, x, v)*, wobei* $a \in \{s, u, i, d\}$ *das Symbol für eine Aktion (Select, Update, Insert oder Delete),* x *für einen XPath-Ausdruck im Sinne von Def. 3, also für das in der Aktion relevante Objekt, und* v *optional für einen zu schreibenden Wert (im Falle eines Schreib- bzw. Einfügezugriffs) steht.*

2.6.1. Lesezugriff

Die XPath-Ausdrücke werden – jeweils abhängig von der Aktion, die auf der Datenbank ausgeführt werden soll – in SQL-Ausdrücke übersetzt. Hierbei ist die Aktion s, ein Lesezugriff (SELECT), die am häufigsten verwendeten Aktion (in Web-Applikationen werden Lesezugriffe 10–20 Mal häufiger verwendet als Schreibzugriffe). Für einen Ausdruck x und ein Hilfsdokument H wird der Ausdruck $x = \texttt{/}K_0p_0\texttt{/} \ldots \texttt{/}K_np_n\texttt{/@}a_1, \ldots, \texttt{@}a_j$ in einen SELECT mittels einer Verbundoperation über die Knotentests $K_i = \{k_i\}$ berechnet, d.h. in Relationenalgebra ausgedrückt als

$$\pi[\texttt{@}a_1, \ldots, \texttt{@}a_j] \overset{n}{\underset{i=1}{\bowtie}}_{\theta_i} (\sigma[p_i](R_i))$$

wobei

$$R_i = \begin{cases} lab(k_i) & \text{falls } att(k_i, \texttt{'rel'}) \text{ undefiniert} \\ att(k_i, \texttt{'rel'}) & \text{sonst} \end{cases} \tag{2.1}$$

und

$$\theta_i = (R_{i-1}.a_1 = R_i.\alpha_1 \wedge \ldots \wedge R_{i-1}.a_k = R_i.\alpha_k)$$

falls $att(k_i, \text{`}join\_condition\text{`})$ der Form

$$\texttt{\%parent.}a_1 \texttt{ = \%this.}\alpha_1\texttt{, ...,\%parent.}a_k \texttt{ = \%this.}\alpha_k$$

ist. Die Verbundoperation stellt hierbei sicher, dass die durch den XPath-Ausdruck beschriebene Enthaltensein-Beziehung durch die Daten in der Datenbank tatsächlich gegeben ist. Die entsprechenden Bedingungen des θ-Verbundes sind im Hilfsdokument festgelegt, und sorgen immer dafür, dass in jedem „Schritt“ des Verbundes die Relation R_{i-1} (*parent*) korrekt mit der Relation R_i (*this*) verknüpft wird.

Beispiel 4 (SELECT in SQL)**.** Sei die Datenbankoperation gerade

(‘s‘, ‘`/schema/mandant[id = 1]/analyse[id = 2]/bewertung[pid = 1 && bid = 2]/@wert`‘)

Dieser wird in SQL als SELECT – Bezug nehmend auf die Hilfsstruktur aus Abb. 2.7 – wie folgt ausgewertet:

```
SELECT r3.wert FROM mandant AS r1
               JOIN analyse AS r2 ON (r1.id = r2.mid)
               JOIN bewertung AS r3 ON (r2.id = r3.bid)
               WHERE (r1.id = 1)
                  AND (r2.id = 2)
                  AND (r3.pid = 1 AND r3.bid = 2)
```

Im Falle der Instantiierung aus Abb. 2.3 liefert dies gerade den atomaren Wert `100`.

2.6.2. Schreibzugriff

Ein Schreibzugriff (UPDATE) kann mittels der vorgestellten Syntax immer nur auf einen atomaren Wert erfolgen, d.h. soll ein gesamtes Tupel t aktualisiert werden, so sind entsprechend der Stelligkeit des Tupels mehrere UPDATE-Operationen notwendig.[13] Sei $(\text{`}u\text{`}, x, v)$ die UPDATE-Aktion, wobei $x = K_0p_0\texttt{/} \ \ldots \ \texttt{/}K_np_n\texttt{/@}a$, sowie R_i wie in Gleichung 2.1 und v der zu schreibende Wert gegeben.

[13] Aus Performanz-Gründen ist jedoch auch ein UPDATE auf ein ganzes Tupel möglich.

Diese Aktion wird dann in den folgenden SQL-Ausdruck übersetzt:[14]

`UPDATE` R_n `SET` $a = v$ `WHERE` p_n

Von dem Ausdruck x werden also lediglich der n'te Knotentest sowie die letzten Prädikate berücksichtigt. Die Prädikate werden nicht in ihrer ursprünglichen Form verwendet, sondern durch äquivalente SQL-Prädikate ausgetauscht. Um sicherzustellen, dass dieser Ausdruck gemäß der vorgestellten XPath-Semantik gültig ist, also die Knotentests und Prädikate $K_0p_0/\ldots/K_{n-1}p_{n-1}$ auch korrekt sind, kann ein entsprechender SELECT-Ausdruck abgesetzt werden, und somit überprüft werden, ob das Ergebnis der Anfrage einen Wert liefert.

2.6.3. Einfügen von Datensätzen

Eine Einfüge-Operation (INSERT) ($'i', x, t$), wobei $t = (t_1, \ldots, t_n)$ ein Tupel über den Attributen $(a_1, \ldots, a_n)$ ist, erfordert, dass der XPath-Ausdruck x in der Form $K_0p_0/\ldots/K_n$ vorliegt, d.h. dass p_n leer ist (der XPath-Ausdruck demnach eine Relation lokalisiert). Dann ist der Ausdruck für die INSERT-Operation in SQL wie folgt:

`INSERT INTO` R_n `WHERE` p_n

2.6.4. Löschen von Datensätzen

Eine Löschoperation (DELETE) ($'d', x$) bezieht sich immer auf ein- oder mehrere Tupel einer Relation, daher muss hier der Ausdruck x der Form $K_0p_0/\ldots/K_np_n$ vorliegen. In Analogie zur INSERT-Operation werden lediglich der Knotentest K_n sowie die Prädikate p_n verwendet, wobei die Prädikate analog übersetzt werden, um den entsprechenden SQL-Ausdruck zu gewinnen:

`DELETE FROM` R_n `WHERE` p_n

2.7. Zwischenfazit

Mit dem vorgestellten Datenmodell sowie dem Datenzugriff mittels XPath sind nun alle Werkzeuge gegeben, um relationale Datenbanken wie XML-Dokumente

[14] Ein entsprechender (einfacher) Ausdruck in relationaler Algebra existiert hierfür nicht, ebensowenig wie für die im folgenden erörterten Operationen für das Löschen und das Einfügen von Daten.

anzusprechen und zu manipulieren. Objekte der Datenbank werden mittels XPath-Ausdrücken identifiziert. Obwohl die zusätzliche Schicht zwischen Datenbank und Client Performanz kostet, sprechen dennoch einige Faktoren dafür:

- ✓ Ein Ausdruck, welcher der zuvor vorgeschlagenen XPath-Syntax entspricht, stellt, wenn er normalisiert wird, eine im Wesentlichen eindeutige String-Repräsentation einer Adressierung eines Objektes dar.[15]
- ✓ Wenn ein Datenbankwert zuerst ausgelesen werden muss, vom Benutzer verändert wird und dann zurückgeschrieben wird, bräuchte man mittels SQL zunächst ein SELECT-Statement, gefolgt von einem UPDATE-Statement. Mittels XPath steht jedoch für jedes Objekt genau ein Ausdruck zur Verfügung, aus dem die dazugehörigen SQL-Statements für das Lesen und Schreiben automatisch generiert werden können.
- ✓ Mittels einer solchen Abstraktionsschicht können neben einer Datenbank weitere Datenquellen, beispielsweise native XML-Dokumente oder gewöhnliche Dateisysteme, angesprochen werden.

Es muss noch bemerkt werden, dass das hier vorgestellte Datenzugriffsmodell lediglich eine Teilmenge der Datenzugriffe erlaubt, die üblicherweise mittels SQL oder relationaler Algebra ausgedrückt werden können. Nicht unterstützt werden beispielsweise Verbundausdrücke (allerdings lassen sich solche in XPath 1.0 gemäß W3C ebenfalls nicht ausdrücken), und Unterabfragen (Sub-SELECTS). Für den Hintergrund, einfache Datenstrukturen bearbeiten zu wollen, ist dies zunächst ausreichend. Sollen jedoch komplexere Daten mit ihren Zusammenhängen bearbeitet werden können, wären zumindest Verbundanfragen wünschenswert.

In dem folgenden Kapitel wird TransForm, eine Schnittstelle zwischen Web-Anwendungen und Datenhaltungseinrichtungen, vorgestellt, die mittels XPath-Ausdrücken auf Datenbanken zugreifen kann.

[15] Eine Normalisierung bedeutet in diesem Kontext, dass anführende und abschließende Slashes (/) und Leerzeichen entfernt, und die jeweiligen Prädikatsausdrücke in konjunktive Normalform gebracht werden. So wird, wie in Kapitel 3 beschrieben, sogar die Eindeutigkeit dieser Ausdrücke erreicht.

Kapitel 3.

TransForm

Dieses Kapitel soll einen Überblick über TransForm geben, einen neuartigen Ansatz für die Interaktion zwischen Benutzer und Datenbanken in webbasierten Applikationen. TransForm kann mittels XHTML-Markup direkt in Webseiten eingebettet werden und garantiert sowohl ACID-Eigenschaften für Transaktionen als auch serialisierbare Schedules.

Zuerst werden theoretische Grundlagen und die Architektur von TransForm vorgestellt sowie das neue Modell zur klassischen Client-Server-Architektur kontrastiert. Ferner werden die vorhandene Implementierung sowohl auf Client- als auch auf Serverseite dokumentiert sowie deren Arbeitsabläufe und Spezifikationen detailliert beschrieben.

3.1. Einführung und Überblick

TransForm[1] [7] ist ein neuartiges Modell, Daten in Formularen auf Webseiten mittels XPath an Datenhaltungsmöglichkeiten anzubinden und ein Interface für die Darstellung und Bearbeitung dieser bereitzustellen, indem spezielle Tags in Webseiten integriert werden. Durch ein Client-Programm werden diese Tags ausgelesen, als Anfragen an einen Server gesendet und die Antworten in geeigneter Weise in die Webseite eingefügt. Durch moderne Browser-Technologien können Aktionen des Clients auf Serverseite in einer Transaktion gebündelt werden, indem alle Anfragen serverseitig zwischengespeichert und erst auf Interaktion des Benutzers in der Datenhaltung dauerhaft gespeichert werden.

Das Modell wurde 2006 beschrieben [7] und ein Prototyp im Sinne eines *„Proof of Concept“* (dt. *Machbarkeitsbeweis*) implementiert, um die generelle Realisierbarkeit der dort beschriebenen Modelle in ihrer Kernfunktionalität nachzuweisen. Bevor diese Diplomarbeit geschrieben wurde, existierten andere Arbeiten und dazugehö-

[1] Der Name leitet sich aus den Worten „Trans“aktion“ und „Form“ular“ ab.

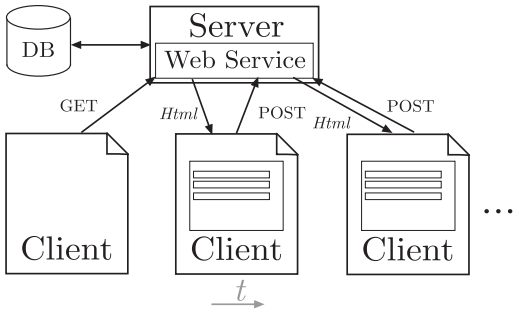


Abb. 3.1.: Klassische Verarbeitung eines Web-Formulars[2]

rige Implementierungen von TransForm [12, 20], die jedoch nicht im Kontext einer kommerziellen Web-Anwendung ansetzen.

In dieser Diplomarbeit wird TransForm im Kontext einer Web-Anwendung und einer SQL-Datenbank implementiert, wobei die Komponenten und das Design der Applikation generell beibehalten wurden; in einigen Fällen weicht die hier beschriebene Spezifikation aus praktikablen Gründen von der des Originals ab, oder musste aufgrund von Anpassungen an diesen Kontext ergänzt werden. Ferner ergaben sich während der Implementierung anhand der konkreten Anwendung einige problematische Aspekte, die separat gelöst werden mussten (beispielsweise die Repräsentation der XPath-Ausdrücke).

Wie bereits in der Einleitung angedeutet, entsteht durch die klassische Verarbeitung von Benutzereingaben – dargestellt in Abb. 3.1 – ein Zyklus zwischen Client und Server, der in der Regel mehrere nacheinander durchgeführte, unzusammenhängende Transaktionen umfasst. TransForm bricht diesen Zyklus auf, und ermöglicht, diese Transaktionen in einer zusammenhängenden Transaktion auszuführen, ohne dass die Notwendigkeit eines zusätzlichen Transaktionsservers besteht. Hierzu implementiert TransForm eigene Browser-Tags (siehe Abschnitt 3.4), um Eingabemasken in Web-Formularen bereitzustellen. Diese spezifizieren die Parameter der verwendeten Server, die Anbindung von Formularelementen an Datenquellen und weitere Metainformationen und Verbeitungsdirektiven.

In dem folgenden Kapitel soll zunächst die generelle Architektur von TransForm beschrieben und ein Überblick über den Anwendungsablauf gegeben werden. Ferner wird detailliert beschrieben, wie die Komponenten implementiert wurden, sowie auf Probleme und Abweichungen von der Spezifikation nach [7] gegeben.

3.1.1. Architektur

TransForm gliedert sich in zwei Komponenten: Den Client und den Server.

[2]Diese Abbildung ist, ebenso wie Abb. 3.2, an die Abbildung aus [7], S. 2, angelehnt.

- Die Aufgabe des **Clients** ist es, die TransForm-basierten Elemente einer Webseite und die Ereignisse der Interaktion des Benutzers zu erkennen, hierzu korrespondierende Anfragen an den Server zu bündeln und die entsprechenden Antworten des Servers in den Prozess der Interaktion zurückzuführen. Dazu gehören die Erkennung der Browser-Tags, die Formulierung von entsprechenden Anfragen an eine Datenquelle, die Re-Transformation der entsprechenden Anfrageergebnisse sowie zusätzlich die Anforderung einer Transaktions-ID.

- Der **Server** übernimmt die Rolle eines Koordinators der Datenzugriffe. Durch ihn erfolgt die Vergabe von eindeutigen Transaktions-IDs und das Bereitstellen einer Schnittstelle zu darunterliegenden Datenobjekten, wie Datenbanken, XML-Dokumenten oder Dateisystemen. Alle Datenzugriffe werden mit der dazugehörigen Transaktions-ID in einer internen Tabelle protokolliert und auf Konflikte überprüft, bis der Benutzer entweder einen COMMIT durchführt, also anfordert, die Änderungen in der korrespondierenden Datenquelle persistent abzuspeichern, oder seine Sitzung im Falle eines ABORTS abbricht.

3.1.2. Programmiersprachen der Implementierung

Während der Server in einer (relativ frei wählbaren) Programmiersprache wie Perl, Java Server Pages oder PHP implementiert werden kann (wobei in dieser Arbeit PHP für die konkrete Implementation gewählt wurde), bleibt auf Seiten des Clients lediglich JavaScript für die Realisierung der oben beschriebenen Anforderungen, da einzig diese Sprache sowie die dazugehörigen Erweiterungen auf nahezu allen modernen Browsern ausreichend implementiert und verfügbar ist[3].

Ferner verwendet der TransForm-Client die *AJAX-Technologie* [5], um Anfragen des Clients abzusenden und somit Benutzereingaben auf Serverseite auszulesen. AJAX ist ein Akronym für die Wortfolge „*Asynchronous Javascript and XML*“ und bezeichnet ein Konzept der asynchronen Datenübertragung zwischen Client und Server, das es ermöglicht, innerhalb einer HTML-Seite eine HTTP-Anfrage durchzuführen, ohne die Seite komplett neu laden zu müssen.

Prinzipiell ist diese Funktionalität auch mit alten Browsern mittels „unsichtbarer Frames“ simulierbar, jedoch besitzen moderne Browser ein JavaScript-Objekt, mit dem gezielt gewisse Teile einer HTML-Seite oder auch reine Nutzdaten sukzessiv bei Bedarf nachgeladen bzw. versendet werden können, wobei das allgemeine Austauschformat der Kommunikation üblicherweise XML bzw. XHTML ist. Da diese Technik eine elegante und saubere Lösung dessen darstellt, was benötigt wird, wurde sie in dieser Implementierung verwendet. Dies ist ebenfalls in [7] vorgesehen.

[3] insofern nicht auf Anwendungen Dritter (Flash, Java Applets) zurückgegriffen werden soll

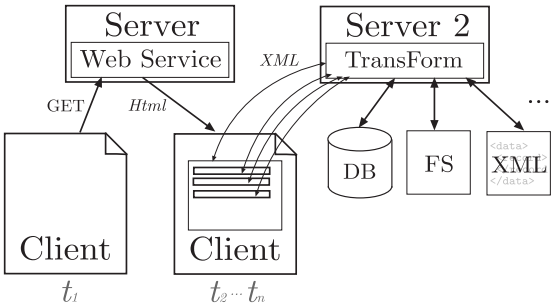


Abb. 3.2.: Verarbeitung eines Formulars mit TransForm

3.1.3. Anwendungsablauf

TransForm macht sich die Möglichkeit zunutze, die Webseite genau einmal komplett beim Server anzufragen, weitere Anfragen werden dann mittels AJAX-Aufrufen realisiert. Dieses Verfahren hat mehrere Vorteile:

- ✓ In jedem Prozess der Server-Client-Interaktion wird nicht die gesamte Webseite (Formulare, Daten, welche der Nutzer eingegeben hat, andere Elemente der Webseite) übertragen. So wird viel unnötiger „Overhead“ vermieden.
- ✓ Eine Transaktion der Datenbank (oder anderen Datenhaltungssystemen) ist nicht mehr auf die Laufzeit eines Skriptes auf dem Server limitiert, sondern wird, durch die Protokollierung aller Zugriffe, über die Dauer der gesamten Sitzung möglich.
- ✓ Durch die Rolle des Servers „als Koordinator“ wird ein effektiver Mehrbenutzerbetrieb ermöglicht.

Im Gegensatz zur klassischen Verarbeitung von Formularen wird die TransForm-basierte Webseite nur einmal durch den Client angefordert und zurückgegeben; durch ein JavaScript-Programm werden dann Anfragen an einen (möglicherweise vom Webserver verschiedenen) Server durchgeführt, welcher den TransForm-Service implementiert. Der genaue Ablauf (dargestellt in Abbildung 3.2) ist nach [7] folgendermaßen:

- Der in JavaScript implementierte TransForm Client (im Folgenden auch als `transform.js` bezeichnet) untersucht den Inhalt der eigenen Webseite auf TransForm-Formulare und fordert für jeden TransForm-Server, identifiziert durch seine Service-URL im jeweiligen Formular, eine Transaktions-ID an, falls noch keine ID für diesen Server vorliegt.

- Liegen alle Transaktions-IDs vor (d.h. eine angeforderte Transaktions-ID wurde zurückgegeben oder ist aus einer vorherigen Anfrage noch gültig), werden im Client für jedes TransForm-Formular alle Tags ausgelesen, die benötigten Daten bestimmt (Formularelemente, Typen und Datenbankobjekte) und mittels AJAX-Request eine Anfrage an den Server gesendet.

- Wenn alle Antworten der Server vorliegen, werden diese XML-Daten geparsed und in der Regel alle TransForm-spezifischen Tags in gewöhnliche HTML-Elemente umgewandelt; entsprechende Trigger-Funktionen erkennen, sobald sich auf dem Client ein Wert verändert oder eine Aktion durchgeführt wird, und teilen dies abermals mittels AJAX-Request dem TransForm-Server mit.

Der entscheidende Vorteil ist, dass eine Transaktions-ID im Programm solange wiederverwendet werden kann, wie die Seite im Browser noch geladen ist. Mittels DOM-Technologie können einzelne Bereiche der Webseite ausgetauscht werden, die dann ebenfalls durch TransForm verarbeitet werden können.

3.1.4. Datenzugriff

Um eine Adressierung von Daten aus einer Datenquelle vornehmen zu können, werden XPath-Ausdrücke verwendet [7]. Exemplarisch wird hier eine Implementierung von TransForm entwickelt, die sich auf die in Kapitel 2 vorgestellten XPath-Ausdrücke bezieht. Diese werden auf Serverseite in SQL-Ausdrücke übersetzt und auf eine relationale Datenbank angewendet. Das zum Einsatz kommende Datenbanksystem ist PostgreSQL 8.2.3 [16], da es frei verfügbar ist und in seiner Implementierung den ISO-Standards SQL-92, SQL:1999 und SQL:2003 am meisten entspricht. Generell lassen sich jedoch auch mit leichten Modifikationen andere Datenbanken, oder aber Archive von XML-Dokumenten oder reine Dateisysteme mittels dieser Technik adressieren. In Kapitel 5 werden entsprechende Erweiterungen diskutiert.

3.2. Implementierung des Clients

In diesem Abschnitt soll genauer auf die konkrete Implementierung mittels JavaScript, die dabei aufgetauchten Probleme und die hierzu erarbeiteten Lösungen eingegangen werden. In [7] ist lediglich grob spezifiziert, dass der Client mittels eines JavaScript-Programmes den Start einer Transaktion initiieren und Werte für Formularelemente aus einer Datenbank anfragen muss, wobei Trigger-Funktionen Veränderungen dieser Werte erkennen und diese in die Datenbank zurückschreiben.

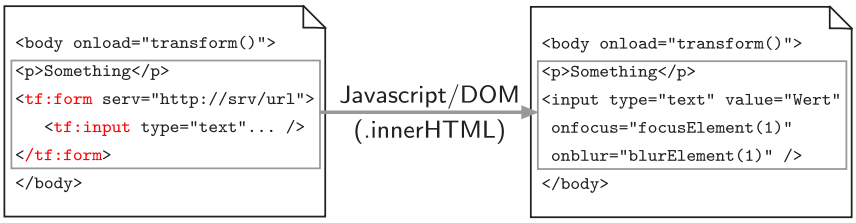


Abb. 3.3.: Aufgabe des Client-Javascript-Programms

3.2.1. Client-Architektur

Um auf Clientseite TransForm in eine Webseite integrieren zu können, wird das JavaScript-Programm `transform.js` benötigt, welches die nötigen Anfragen an den TransForm-Server absetzt, Antworten auf solche Anfragen koordiniert und vom Server zurückgegebenen Inhalt in die Webseite integriert. Die Datei `transform.js` wird hierbei in den Kopf der HTML-Seite eingebunden, auf der TransForm zum Einsatz kommen soll.

Die genauen Anforderungen an dieses Programm lauten:

1. Es muss – entweder nach dem Ladevorgang der eigenen Seite, oder aber durch Interaktionen des Benutzers – Teile der Webseite auf TransForm-spezifische Browser-Tags, wie in Abschnitt 3.4 beschrieben, durchsuchen.

2. Alle erkannten TransForm-Formulare müssen daraufhin ihre jeweiligen Server kontaktieren und eine Transaktions-ID anfordern.

3. Wenn die Transaktions-IDs vorliegen, müssen die jeweiligen Daten auf den Servern angefragt werden, sowie deren Antworten gespeichert werden.

4. Liegen die Antworten der zuvor abgesetzten Anfragen vor, muss das Programm die Inhalte auf der eigenen Webseite austauschen und durch gültiges XHTML ersetzen.

In der Regel wird im `<body>`-Tag der geladenen TransForm-Webseite mittels `onload="transform()"` die Client-Hauptfunktion ausgeführt, welche die vier genannten Aufgaben erledigt. Diese kann – bei Bedarf – auf später nachgeladene Teile der Webseite angewendet werden, um (möglicherweise mittels AJAX-Request) nachgeladene Inhalte mittels TransForm zu verarbeiten.

3.2.2. Erkennung und Auswertung der Formulare und Tags

Um die Erkennung der Browser-Tags zu bewerkstelligen, wird der Inhalt des DOM-Knotens, auf den die Funktion `transform(`$\langle$*Knoten*$\rangle$`)` angewendet wird, mittels der

Funktion `innerHTML` ausgelesen. Wird kein Knoten übergeben, so wird der Inhalt des `<body>`-Tags der geladenen Seite angenommen. Der nun in einer JavaScript-Variable vorliegende HTML-Inhalt wird mit einem JavaScript-SAX-Parser ausgewertet, um alle in diesem String enthaltenen XML-Tags zu erhalten und auf TransForm-spezifische Tags untersuchen zu können (Abb. 3.3).[4]

Diese Lösung ist im Wesentlichen mit mehr Nach- als Vorteilen versehen:

✓ Der SAX-Parser ist in JavaScript implementiert und somit auf jedem Browser lauffähig, wenn JavaScript unterstützt wird.

✗ Jeder Browser implementiert Funktionen zum Verarbeiten von XML bzw. XHTML. Einige Browser stellen derartige Funktionen in JavaScript bereit (beispielsweise der Internet-Explorer mittels eines ActiveX-Objektes), diese sind jedoch unflexibel und nicht auf allen Plattformen implementiert, sowie von den Sicherheits- und Konfigurationseinstellungen des Browsers abhängig. Ferner erwarten diese als Eingabe keine Zeichenkette, sondern eine URL. Daher muss hier – der mangelnden Browserunterstütung wegen – zusätzliche Funktionalität implementiert werden, anstatt auf vorhandene Funktionsbibliotheken des Browsers zugreifen zu können.

✗ Ein SAX-Parser erwartet normalerweise nach Spezifikation gültiges XML bzw. XHTML. Wird der Inhalt des `<body>`-Tags mittels

```
var contents = document.getElementsByTagName('body')[0].innerHTML
```

in eine JavaScript-Variable übertragen, so ist es abhängig von den jeweiligen Browsern, ob `innerHTML` auch gültiges XHTML zurückliefert; so bricht beispielsweise der Internet-Explorer[5] den Standard, indem er Attributwerte nicht in Hochkommata einschließt, wenn dies nicht zwingend notwendig ist. Der SAX-Parser muss daher angepasst werden, um auf möglichst vielen Browsern lauffähig zu sein: seine validierenden Eigenschaften werden vollständig ausgeschaltet, sowie Mechanismen hinzugefügt, auch vom Internet Explorer zurückgegebenes `innerHTML` verarbeiten zu können.

Prinzipiell wäre die Anwendung einer DOM-Traversierungsfunktion[6] auf den Ausgangsknoten (wie in Abb 3.5) eine weitere Lösung. Ausgehend von einem Startknoten

[4] In [7] wird aufgeführt, dass die Tags anhand ihrer DOM-Repräsentation identifiziert werden müssen. Diese Lösung erweist sich jedoch Aufgrund mangelnder Browser-Unterstützung als nicht praktikabel. Um die Erkennung zu realisieren, wurde daher ein SAX-Parser verwendet und angepasst. Näheres zu dem verwendeten SAX-Parser findet sich im Anhang.

[5] Version 6.0

[6] DOM = *„Document-Object-Model“*, ein Ansatz, Elemente von XHTML-Bäumen objektorientiert zu adressieren

würden rekursiv alle Kindknoten durchsucht und auf entsprechende TransForm-Tags geprüft. Hier erweist sich ebenfalls die Implementierung des Internet Explorers als problematisch, da dort für unbekannte XHTML-Tags[7] die Funktion `innerHTML` nicht verfügbar ist[8]. Ferner repräsentieren bei einer Traversierung ihm unbekannter Tags nicht nur öffnende, sondern auch abschließende Tags einen eigenen DOM-Knoten, was vollkommen konträr zu dem Document-Object-Modell ist. Des Weiteren werden durch die TransForm-Tags in den zurückgegebenen `innerHTML`-Inhalten falsche *Processing-Instructions* eingefügt, die der SAX-Parser nicht verarbeiten kann. Daher wird von dieser Lösung abgesehen und der SAX-Parser angepasst: Dieser interpretiert nun nicht-abgeschlossene Tags, Attribute, deren Werte nicht in Hochkommata eingeschlossen sind, ignoriert Processing-Instructions und führt für einige Entitäten keine Übersetzung durch, anstatt mit einem Fehler abzubrechen. Ziel dieser „Aufweichung“ der XHTML-Kriterien ist es, zumindest die beiden weit verbreiteten Browser, Internet Explorer und Mozilla Firefox, unterstützen zu können.

Jedoch scheint auch Mozilla Firefox Probleme mit der `innerHTML`-Funktion von benutzerdefinierten Tags zu haben; so werden Tags künstlich als ineinander geschachtelt interpretiert (siehe Abb. 3.4), bzw. selbst-abschließende Tags nicht als solche erkannt. Dies kann mitunter zu sonderbaren Resultaten führen.

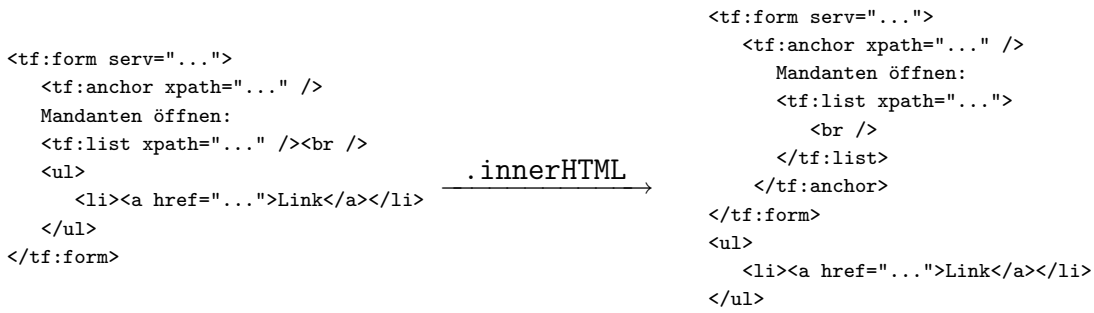


Abb. 3.4.: XML-Inhalt und dazugehöriges `innerHTML` von Mozilla Firefox

So werden nun, für einen gegebenen DOM-Knoten, alle in ihm enthaltenen TransForm-spezifischen Tags der Form `<tf:`⟨*Tagname*⟩`.../>` ausgelesen und ein Array mit dem jeweiligen Tag-Name, seinen Attribut-Werte-Paaren und die Offsetposition innerhalb des Original-Strings zurückgegeben. Zunächst sind für die Identifizierung der Formulare und ihrer jeweiligen Server lediglich die Tags der Form `<tf:form .../>` von Relevanz.

[7] d.h. solchen, die nicht in [23] spezifiziert sind
[8] da diese Tags für den IE kein gültiges XHTML darstellen

```
function traverseDOMTree(currentElement) {
    var tagName=currentElement.tagName;
    if (tagName.substr(0,3)=='TF:'
        || tagName.substr(0,4)=='/TF:') {
        transFormElements.push(currentElement);
    }
    var i=0;
    while (Child = currentElement.childNodes[i]) {
        traverseDOMTree(Child);
        i++;
    }
}
```

Abb. 3.5.: Traversierung eines XHTML-Dokumentes mittels DOM

3.2.3. Anfordern der Transaktions-IDs

Für jedes TransForm-Formular, das mittels eines Tags der Form `<tf:form...>`[9] eingeleitet wird, wird die durch den Parameter `serv="..."` spezifizierte URL als Service-URL gespeichert und geprüft, ob für sie schon eine Transaktions-ID vorliegt. Trifft dies zu, wird diese für den folgenden Schritt verwendet. Trifft dies nicht zu, wird mittels AJAX-Request ein BEGIN-Request gestartet (Abb. 3.6). Für diesen Schritt werden mittels POST alle notwendigen Daten an die URL gesendet. Beim ersten BEGIN sollten Authentifizierungsinformationen beigefügt werden. Die Parameter des POST-Strings ergeben sich aus Tabelle 3.1.[10]

Parameter	Beschreibung
`a=b`	Aktion ist BEGIN
`obj=`⟨*Index*⟩	Index des aufgerufenen Formularobjektes (intern verwendet)
`objURL=`⟨*URL*⟩	URL des aufgerufenen Servers[11]
`u=`⟨*Name*⟩	Benutzername
`p=`⟨*Pass*⟩	Passwort als MD5-gehashter 32-Oktett-Wert

Tab. 3.1.: Parameter beim Aufruf von BEGIN

Der Server liefert auf einen solchen Request folgende XML-Daten als Rückgabe:

[9] Es ist möglich, in einem TransForm-Dokument mehrere Formulare mit unterschiedlichen Server-URLs zu spezifizieren.

[10] Das Format der Parameter unterscheidet sich zu dem in [7] vorgegebenen Format. Näheres hierzu findet sich in Abschnitt 3.5.

[11] dies dient der Überprüfung auf Serverseite, ob der Client den richtigen Server aufzurufen beabsichtigt.

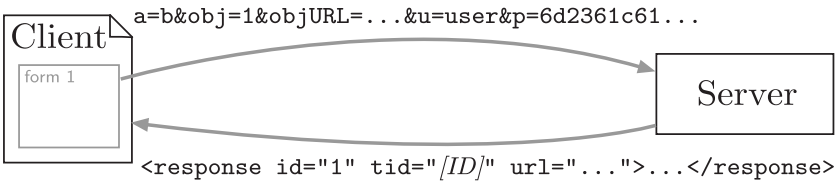


Abb. 3.6.: BEGIN-Request mit Authentifizierung und Antwort

```
<response id="0" tid="4" url="http://www.example.com/server/tf.php">
    <responseCode>201</responseCode>
</response>
```

Auf Clientseite werden die Transaktions-IDs in Hashes der Form $\langle URL \rangle \mapsto \langle ID \rangle$ abgespeichert. Die vom Server zurückgegebenen Statuscodes sind ähnlich den HTTP-Statuscodes und in Anhang A.1 aufgeführt. Der Wert zu `id` korrespondiert zu dem in Tabelle 3.1 aufgeführten Wert für $\langle Index \rangle$ und wird im Client verwendet, um die Antwort dem ausgeführten Request zuzuordnen. Eine solche Lösung ist notwendig, da bei dem asynchronen Absetzen mehrerer Requests keine Vorhersage darüber getroffen werden kann, welcher Request zuerst erfolgreich terminiert und wann die dazu korrespondierende Antwort vorliegt.

3.2.4. Auslesen der Datenbankdaten

Wenn die Transaktions-IDs aller in der Seite aufgefundenen Formulare vorliegen, wird jeweils ein weiterer Aufruf pro Formular getätigt, der für jedes `<tf:.../>`-Tag innerhalb eines Formulars Daten enthält. Diese werden in Form eines POST-Strings mit den in Tabelle 3.2 aufgeführten Parametern[12] abermals mittels AJAX-Request an die entsprechende URL gesendet.

[12]Die Notation einer Variablen `var` mit eckigen Klammern `[]` indiziert, dass diese für jedes TransForm-Element an den POST-String angehängt werden; PHP interpretiert diesen automatisch als Array.

Parameter	Beschreibung
`obj=`⟨*Index*⟩	Index des aufgerufenen Formulars (intern verwendet)
`tid=`⟨*ID*⟩	Zuvor erhaltene Transaktions-ID für das Formular
`a[]=r`	Aktion ist READ
`i[]=`⟨*i*⟩	Index des Tags (intern verwendet)
`t[]=`⟨*Typ*⟩	Typ des Tags, d.h. `input` im Fall von `<tf:input .../>`
`x[]=`⟨*XPath*⟩	XPath für das Tag
`tpl[]=`⟨*Template*⟩	Serverseitig mit Daten aufzurufendes Template

Tab. 3.2.: Parameter beim Aufruf eines READ-Requests

Die Antwort auf einen Request für ein Formular sieht typischerweise wie folgt aus:

```
<response id="0" tid="4">
    <responsecode>201</responsecode>
    <responsecontent>
        <output index="1">...</output>
        ...
        <output index="n">...</output>
    </responsecontent>
</response>
```

Für jedes `<tf:.../>`-Tag wird (in Analogie zu den Formularen) intern ein Zähler, hier *Index* genannt, gehalten. Die Indizes in der entsprechenden XML-Rückgabe korrespondieren zu den Indizes einer globalen JavaScript-Variable, in welcher clientseitig alle Rückgaben verwaltet werden. Die XML-Antwort wird nun clientseitig ebenfalls mittels SAX-Parser verarbeitet, um die entsprechenden Rückgabedaten – einfacher Datenbank-Inhalt oder aber XML-Daten – zu erhalten. Diese Ausgabe ist eingefasst in die öffnenden und schließenden `<output>`-Tags. Clientseitig ist eine XSLT-Schnittstelle vorhanden, die weitere Transformationen erlaubt. Weiterhin kann serverseitig eine Verarbeitung durch sogenannte Templates stattfinden. Diese werden im Anhang, Abschnitt A.2, beschrieben.

3.2.5. Übersetzen und Austauschen von Inhalten

Liegen die Antworten aller Formulare vor, kann damit begonnen werden, die entsprechenden Daten innerhalb der Webseite auszutauschen. Das Austauschen wird mittels bei der Identifizierung der TransForm-Tags berechneten String-Positionen bewerkstelligt. Diese String-Positionen werden herangezogen, um die TransForm-Tags sukzessive in korrektes HTML zu übersetzen. Hierbei wird zwischen *einfachen*,

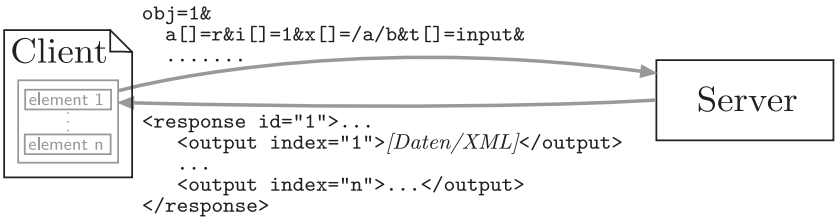


Abb. 3.7.: READ-Request mit Antwort

komplexen und solchen TransForm-Tags unterschieden, deren zurückgelieferte Daten auf Clientseite mittels eines XSLT-Prozessors weiterverarbeitet werden.

- Als **einfache Tags** werden Tags bezeichnet, die atomare Datenbankwerte erwarten und bearbeiten können, in der Regel also XHTML-Eingabefelder [23, 22] zur Bearbeitung von Text (`<input>`, `<textarea>`, `<select>` u.A.), und Transform-Steuertags (beispielsweise Buttons oder Ankerpunkte, beschrieben in Abschnitt 3.5). Diese werden direkt – zusammen mit ihren Inhalten – in XHTML-Elemente umgewandelt. Zu den Elementen werden zusätzliche *Trigger-Funktionen* hinzugefügt, welche die Interaktionen des Benutzers verarbeiten und im folgenden Abschnitt näher beschrieben werden.

- Als **komplexe Tags** werden Tags bezeichnet, die nicht-atomare Datenbankwerte erwarten, z. B. Listen oder Auswahlfelder (siehe hierzu Abschnitt 3.4.3). Sie erlauben weitere Aktionen, wie Darstellung, Auswahl, Bearbeitung oder Löschen von Inhalten. Komplexe Tags werden durch XSLT oder durch Templates formatiert, folglich sind die vom Server zurückgegebenen Daten bereits in XHTML-Format.

Sind alle Inhalte entweder durch Client oder Server berechnet bzw. mittels XSLT-Schnittstelle weiterverarbeitet, werden diese in den Quellcode eingefügt und mittels `innerHTML` in das Zielobjekt eingefügt.

3.2.6. Aktionen des Benutzers

Nachdem eine Webseite durch den Client „übersetzt“ worden ist, stehen dem Benutzer in der Regel Eingabefelder mit dazugehörigen Daten zur Bearbeitung zur Verfügung. Für einfache Tags, die vom TransForm-Client in XHTML-Elemente umgewandelt worden sind, erkennen so genannte *Trigger-Funktionen* [7] Veränderungen der Werte in den Eingabefeldern. Hierzu sind die Eingabeelemente mit zwei JavaScript-Attributen

`onfocus="focusElement('`⟨*Formular-Index*⟩`','`⟨*Element-Index*⟩`');"`

sowie

`onblur="blurElement(this,'`⟨*Formular-Index*⟩`', '`⟨*Element-Index*⟩`', '`⟨*Anker*⟩`');"`

ausgestattet, um zu erkennen, wann ein Element den Fokus bekommt (mittels `focusElement`) und wann es den Fokus verliert. Im zweiten Fall überprüft die Funktion `blurElement`, ob sich der Wert des Eingabefeldes gegenüber dem zuvor initial eingetragenen bzw. markierten Wert verändert hat, und signalisiert im Fall einer Veränderung mittels AJAX-Request dem Server einen WRITE mit dem neuen Wert.

Im Falle eines durch den Benutzer ausgelösten COMMITS respektive ABORTS wird ebenso verfahren; die Antwort des Servers ist jeweils die Ausgabe eines XML-Dokumentes, welches den Statuscode der zuletzt ausgeführten Aktion enthält (ähnlich der Antwort auf den BEGIN-Request). Für die Aktionen „Einfügen eines Datensatzes“ oder „Löschen“ existieren ebenfalls solche TransForm-Client-Funktionen. Für alle Aktionen können spezielle Handler-Funktionen definiert werden, die im Falle einer Benutzeraktion ausgeführt werden. Diese sind besonders für den Einsatz von TransForm bei der Implementierung von Applikationen von Bedeutung, und werden in Tabelle 3.3 aufgeführt.

Funktionsprototyp	Beschreibung
`insertedHandler(x,p,r)`	Falls ein Insert-Button betätigt wurde, wird diese Funktion mit dem Basisobjekt `x` und den Prädikaten des eingefügten Objekts `p` ausgeführt. Der Parameter `r` enthält in allen Funktionen die XML-Rückgabe des Servers
`deletedHandler(x,p,r)`	Fall sein Objekt gelöscht wurde, wird diese Funktion mit analogen Parametern zur Funktion `insertedHandler` aufgerufen
`committedHandler(r)` `abortedHandler(r)`	Aufzurufende Funktionen im Falle eines erfolgreich ausgeführten COMMIT bzw. ABORT

Tab. 3.3.: TransForm Handlerfunktionen

Ferner sollte sichergestellt sein, dass eine Sitzung im Browser nicht zu lange ohne Aktion bleiben darf, bzw. bei Beendung der Sitzung der Server über einen ABORT informiert wird; dies sollte üblicherweise in der Applikation, die TransForm implementiert, geschehen, indem der Browser mittels `onunload`-Befehl noch eine Aktion an den Server absetzen kann, sobald das Fenster geschlossen wird. Stürzt jedoch der Browser ab, muss der Server den ABORT selbstständig durchführen. Näheres wird hierzu in Abschnitt 3.3.5 geschildert.

3.3. Implementierung des Servers

Im folgenden Abschnitt wird die Implementierung des Serverdienstes vorgestellt, die Unterschiede zur ursprünglichen Spezifikation nach [7] hervorgehoben sowie Gründe für Abweichungen gegeben[13]. Alle beteiligten Komponenten des Servers werden detailliert beschrieben.

3.3.1. Überblick

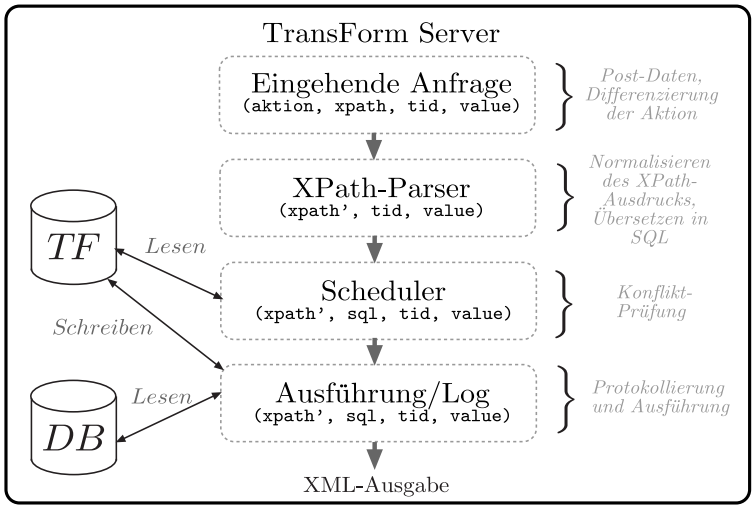


Abb. 3.8.: Komponenten und Ablauf des TransForm-Servers

Der TransForm-Server ist eine aus einer PHP-Datei, `tf.php`, und einer PHP-Klasse, `transform.class.php` bestehende Komponente, die serverseitig die Datenbankzugriffe koordiniert, für die Vergabe der Transaktions-IDs verantwortlich ist, das interne Protokoll der Datenzugriffe (*Logging*) führt, die XPath-Ausdrücke in SQL-Ausdrücke übersetzt und die eigentlichen Abfragen an die Datenbank durchführt.

Der Server dient als Koordinator für die Datenbankzugriffe einzelner Benutzer. Die Anforderungen an den Server sind deshalb:

1. **Datenzugriff und Sicherheit**. Die Authentifizierung des Benutzers und der Datenzugriff sind zentrale Aufgaben des Servers; der Server sorgt dafür, dass Datenquellen mittels XPath-Ausdrücken angesprochen und manipuliert werden können.

2. **Schaffung einer Isolationsebene**. Durch das Isolieren der einzelnen Benutzer müssen die Datenbankzugriffe voneinander so getrennt ablaufen, dass diese

[13]Die Unterschiede werden in Abschnitt 3.5 nochmalig zusammengefasst.

sich nicht gegenseitig beeinträchtigen (Mehrbenutzerbetrieb durch simulierten Einbenutzerbetrieb).

3. **Protokollierung und Logging**. Alle Aktionen müssen protokolliert werden, damit – bis die Daten dauerhaft geschrieben werden – jeder Benutzer lediglich die Ergebnisse eigener Aktionen registriert.

4. **Transaktionsverwaltung und Scheduling**. Aufgrund der letzten zwei Punkte wird eine Transaktionsverwaltung und eine Schedulingstrategie unumgänglich, um einen stabilen Mehrbenutzerbetrieb zu ermöglichen.

Die PHP-Klasse ist für die hier beschriebenen Funktionen zuständig, das PHP-Dokument `tf.php` regelt das Auslesen der Eingaben, die Aufrufe der Klasse und die Generierung von XML-Ausgaben. Das PHP-Dokument wird mit einer Aktion oder einer Folge von Aktionen aufgerufen. Eine serverseitige Aktion ist ein Quadrupel der Form $(action, xpath, tid, value)$, das der Server, wie in Abb. 3.8 skizziert, verarbeitet. Hierbei werden verschiedene Teilkomponenten des Servers aufgerufen bzw. benötigt. Ein Anfrage an den Server durchläuft dabei den folgenden Prozess:

- Alle Anfragen an den Server müssen zunächst auf Korrektheit geprüft und dann anhand ihres Aktions-Identifyers differenziert verarbeitet werden; dies übernimmt eine einfache `switch`-Anweisung in PHP.

- Der **XPath-Parser** übernimmt die Überprüfung auf syntaktische Korrektheit der gegebenen XPath-Ausdrücke und deren Übersetzung in SQL-Anweisungen. Ferner sorgt er dafür, dass die XPath-Ausdrücke normalisiert und somit als Zeichenketten vergleichbar gemacht werden. Einen Überblick über die Funktionsweise des XPath-Parsers liefert Abschnitt 3.3.3 (Objektadressierung und Konflikterkennung).

- Der **Scheduler** bzw. der **Transaktions-Manager** prüft, ob bezüglich einer Aktion Konflikte mit anderen Transaktionen vorliegen und garantiert die ACID-Eingenschaften des Datenzugriffs. Wenn keine Konfikte vorliegen, wird die Aktion der aktuellen Transaktion hinzugefügt. Der Scheduler und die ausgewählte Strategie werden in Abschnitt 3.3.4 genauer betrachtet.

- Wird eine Anfrage als konfliktfrei eingestuft, wird sie entsprechend ausgeführt und protokolliert; da die Datenzugriffe und Manipulationen im Server lokal zwischengespeichert werden und für andere Benutzer nicht sichtbar sind, werden entsprechende Behandlungsmechanismen benötigt, die eine tatsächliche „Isolation“ der Transaktion bewerkstelligen; hierauf wird in Abschnitt 3.3.5 (**Logging und Ausführung**) genauer eingegangen.

Ferner benötigt der Server eine eigene Datenbankinfrastruktur für das interne Log, die im folgenden Abschnitt genauer besprochen wird. Der TransForm Server bzw. das PHP-Dokument `tf.php` kann entweder auf dem gleichen Server ausgeführt werden, oder sich auf einem externen, zweiten Server (wie in Abb. 3.2 illustriert) befinden. Im zweiten Fall müssen die Sicherheitseinstellungen der jeweiligen Browser allerdings so manipuliert werden, dass mittels AJAX Zugriffe auf Hosts stattfinden können, die nicht dem Host entsprechen, auf dem die jeweilige TransForm-Webseite liegt. Die Notwendigkeit der meisten in den folgenden Abschnitten erwähnten Komponenten ergibt sich aus dem in dieser Implementierung verwendeten Daten- und Adressierungsmodell. Diese Komponenten werden in [7] zwar erwähnt, jedoch nicht genauer spezifiziert, da die dort vorgestellten Ansätze sich auf generelle Datenhaltungs- und Adressierungsmöglichkeiten beziehen. Hier wird auf die sich aus dem konkret gewählten Datenmodell ergebenden Probleme eingegangen: Diese umfassen den Objektzugriff mittels XPath, die Eindeutigkeit der Adressierung und die Ausführung und Isolation der einzelnen Objektzugriffe.

3.3.2. Die TransForm-Datenbank

Neben der Datenbankverbindung der eigentlichen Datenbank wird eine zusätzliche Datenbank benötigt, welche die Authentifizierungs- und Protokollierungsinformationen aufnehmen kann, um den Betrieb von TransForm mit ausreichender Performanz und Stabilität zu gewährleisten.[14]

Innerhalb dieser Datenbank befinden sich zwei Relationen; die Relation **transform_auth**, bestehend aus den Attributen *id*, *user* und *password*, sowie die Relation **transform_schedule**. Erstere enthält lediglich die Authentifizierungsinformationen für jeden Benutzer in Form von Benutzerkennung, Benutzername und einem 32-Byte MD5-Hash des Benutzerpasswortes.

Die Relation **transform_schedule** wird für die Protokollierung und Schaffung der Isolationsebene benötigt. Diese besteht im Einzelnen aus folgenden Attributen:

- $tid \in \mathbb{N}$, für die jeweilige Transaktions-ID,
- $action \in \{b, r, w, i, d, c\}$, ein Identifyer der jeweiligen protokollierten Aktion oder eines Zustandes, stehend für Begin, Read, Write, Insert, Delete oder Conflicting,
- $xpath \in \Sigma^*$, der XPath-Ausdruck des Objektes, auf das sich die jeweilige Aktion bezieht,

[14] Genauer gesagt werden zwei Relationen benötigt, welche sich auch in der Ausgangs-Datenbank befinden können; um eine bessere begriffliche Trennung anzudeuten, wird hier von einer eigenen Datenbank gesprochen.

- *value* $\in \Sigma^*$, eine Zeichenkette (den zu schreibenden Wert), falls die Aktion einen WRITE oder einen INSERT darstellt, sowie
- *timestamp*, der Zeitstempel einer ausgeführten Aktion.

Die Instanzen dieser beiden Relationen werden im Folgenden kurz a (Authentifizierungsrelation) und s (Schedulerrelation) genannt, deren Attributmengen A_a bzw. A_s.

Definition 7 (**atomare Datenbankoperation, Transaktion**). *Eine atomare Operation auf der Haupt-Datenbank ist das Tripel* $(action, xpath, value)$*, und eine Transaktion besteht aus einer durch das Attribut timestamp serialisierten Folge solcher atomarer Aktionen, welche dieselbe Transaktions-ID haben.*

Alle Datenzugriffe und Schreiboperationen auf der Hauptdatenbank werden mit der dazugehörigen Transaktions-ID in dieser internen Datenbankrelation protokolliert.

3.3.3. Objektadressierung und Konflikterkennung

In Kapitel 2 wurde bereits erläutert, inwiefern ein Datenbankschema als XML-Dokument adressiert werden kann und XPath-Ausdrücke in SQL-Ausdrücke übersetzt werden können, so dass ein XPath-Ausdruck ein Datenbankobjekt *eindeutig* beschreibt. *Eindeutigkeit* bedeutet in diesem Kontext, dass in dem Datenmodell, welches mittels einer Hilfsstruktur generiert wird, nicht zwei unterschiedliche XPath-Ausdrücke vorkommen, die dasselbe Objekt adressieren. Dies wurde zunächst darüber erreicht, dass die in den XPath-Ausdrücken definierten Knotentests deterministisch sind, da der Name einer Relation als Knoten lediglich einmal im Hilfsdokument eingefügt wird.

Für TransForm wird jedoch ein Gleichheitsbegriff für zwei XPath-Ausdrücke x und x' benötigt, der sich auf eine textuelle Repräsentation der XPath-Ausdrücke bezieht, da innerhalb der TransForm-Datenbank der Zugriff auf Objekte mittels Strings protokolliert wird. Ferner soll für zwei XPath-Ausdrücke eine Vergleichbarkeit dahingehend gewährleistet sein, dass ebenfalls anhand der textuellen Repräsentation entschieden werden kann, ob x' ein *Speziallfall* von x ist, der Ausdruck x' also ein Objekt adressiert, welches Kindobjekt des durch den Ausdruck x adressierten Objektes ist.[15] Dieses Problem ist ähnlich dem *Query-Containment-Problem* [15, 19], wobei für zwei Anfragen q, q' entschieden werden soll, ob für alle Datenbankinstanzen stets $result(q') \subseteq result(q)$ gilt.

[15] Ein geeignetes Beispiel hierfür ist, dass x ein bestimmtes Tupel adressiert, und x' lediglich die Projektion auf ein Attribut genau dieses Tupels darstellt.

Um diesen Problemen begegnen zu können, müssen Ausdrücke zuerst auf eine Normalform gebracht werden. Ein erster Ansatz bezieht sich darauf, die XPath-Ausdrücke anhand ihrer trennenden Slashes und ihrer Struktur mittels regulärem Ausdruck aufzuteilen, die Prädikate abzutrennen, diese zu parsen und in geeigneter Weise alles wieder zusammenzufügen, so dass beispielsweise eine durch fehlende oder wiederholte Leerzeichen (Whitespaces) in einem Prädikakt hervorgerufene textuelle Ungleichheit behoben werden kann.

So sind die Ausdrücke

```
/schema/mandant[(a == 1 || (c == 2 && d == 3))]
```

und

```
/schema/mandant[(a== 1 || (c == 2 && d == 3))]
```

semantisch offenbar identisch. Es herrscht jedoch textuelle Ungleichheit, da im Prädikat des zweiten Ausdruck ein Leerzeichen fehlt. Eine Normalisierung (mittels erneutem Parsen und Zusammensetzen) behebt diese Ungleichheit, allerdings ist der Ausdruck

```
/schema/mandant[(a == 1 || c == 2) && (a == 1 && d == 3))]
```

erneut semantisch äquivalent (da in *konjunktiver Normalform*, KNF), textuell jedoch abermals verschieden. Hierzu produziert der XPath-Parser erst die konjunktive Normalform der gegebenen Prädikate eines Ausdrucks, und sorgt zusätzlich mittels einer Sortierung bei kommutativen Verknüpfungen sowie dem Weglassen überflüssiger Klammern für textuelle Eindeutigkeit.

Im Einzelnen verfährt der Parser wie folgt:

1. Mittels dem regulären Ausdruck[16]

   ```
   /([A-Za-z0-9_@]*::){0,1}([A-Za-z0-9_@]*)(\[[^\]]*\]){0,1}/
   ```

   erhält der Parser ein Array aus allen Teil-Ausdrücken der Form $K_i p_i$, da diese durch Slashes separiert und die Prädikate in eckige Klammern eingefasst sind.

2. Die Prädikate werden geparsed, einem Baum zugeordnet, dieser in KNF transformiert und daraus eine textuelle Repräsentation erzeugt, welche die gewünschte Eindeutigkeit liefert. Dazu werden in kommutativen Verknüpfungen wie beispielsweise `&&` und `||` die einzelnen Operanden zusätzlich sortiert, damit eine textuelle Inäquivalenz von beispielsweise `(id == 1)` und `(1 == id)` ausgeschlossen werden kann.

[16] Dieser reguläre Ausdruck berechnet zusätzlich die Achsen, welche in diesem Fall nicht benötigt werden.

3. Die so entstehenden Prädikatsausdrücke werden dann zusammen mit ihren Knotentests durch Slashes wieder verbunden.

Die Transformation der gemäß Grammatik in Definition 5 gegebenen Ausdrücke in KNF sorgt hierbei dafür, dass die Prädikatsausdrücke eindeutig werden, und somit auch die XPath-Ausdrücke in ihrer textuellen Repräsentation als Strings eindeutig sind:

Behauptung. *Für jeden gemäß Definition 5 gebildeten Ausdruck φ existiert ein Ausdruck φ^N in konjunktiver Normalform: Wenn man die Operatoren* `&&` *mit $\wedge$,* `||` *mit $\vee$ und* `!` *mit $\neg$ identifiziert, und $a_{i,j}$ für die gemäß der Regel* Atom *gebildeten Ausdrücke stehen, so lässt sich φ eindeutig darstellen als Ausdruck φ^N der Form*

$$\left(\bigwedge_{i=1}^{n}\left(\bigvee_{j=1}^{m_i} a_{i,j}\right)\right)$$

Beweis. Mittels Induktion über den Aufbau von φ und Fallunterscheidung. Ist φ atomar (d.h. eine direkte Ableitung der Regel Atom), so ist nichts zu zeigen.

1. Fall: $\varphi = (\varphi_1 \vee \varphi_2)$. Nach Induktionsvoraussetzung existieren die Ausdrücke φ_1^N und φ_2^N in KNF. Seien C_i die Disjunktionen von φ_1^N, und D_i das entsprechende Analogon für φ_2^N, also

   $$\varphi_1^N = \bigwedge_{i=1}^{n} C_i \qquad \text{sowie} \qquad \varphi_2^N = \bigwedge_{j=1}^{m} D_j$$

   so ergibt sich φ^N als Anwendung des Distributivgesetzes zu

   $$\varphi^N = \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{m} (C_i \vee D_j)$$

   in konjunktiver Normalform. Die eindeutige Darstellung ergibt sich aus der Sortierung (und Entfernung von Duplikaten) aus den entsprechenden Konjunktionen und ein Weglassen von überflüssigen Klammerungen.

2. Fall: Ist $\varphi = (\varphi_1 \wedge \varphi_2)$, so existieren nach I.V. die Ausdrücke φ_1^N und φ_2^N in KNF, und $\varphi^N = (\varphi_1^N \wedge \varphi_2^N)$ in KNF. Auch hier werden überflüssige Klammern weggelassen und die Konjunktion sortiert (sowie Duplikate in ihr entfernt).

3. Fall: $\varphi = \neg\varphi_1$. Falls φ_1 atomar, so ist nichts zu zeigen.

   a) Ist $\varphi_1 = \neg\varphi_2$, so ist $\varphi^N = \varphi_2$.

   b) Falls $\varphi_1 = (\varphi_2 \wedge \varphi_3)$ bzw. $\varphi_1 = (\varphi_2 \vee \varphi_3)$, so ist in diesem Fall (nach den De'Morganschen Regeln) $\varphi^N = (\neg\varphi_2^N \vee \neg\varphi_3^N)$ bzw. $\varphi^N = (\neg\varphi_2^N \wedge \neg\varphi_3^N)$, da nach I.V. die Ausdrücke $\neg\varphi_2^N$ und $\neg\varphi_3^N$ existieren und in KNF sind.

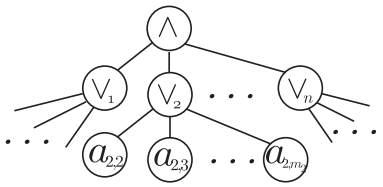


Abb. 3.9.: Baumrepräsentation von φ^N als Konjunktion von Disjunktionen

c) Ist $\varphi_1 = (\varphi_2\ \theta\ \varphi_3)$ für $\theta \in \{=, \neq, <, >, \leq, \geq\}$, so ist $\varphi^N = (\varphi_2\ \theta'\ \varphi_3)$, wobei θ' gerade die Inverse Relation zu θ darstellt (für den Fall $\theta =$ ‘ $<$ ‘ ist $\theta' =$ ‘ $\geq$ ‘).

Dies beendet den Beweis. □

Da der Beweis konstruktiv ist, und folglich mittels einer rekursiven Prozedur der Ausdruck φ^N in KNF erzeugt werden kann, wird jeder Prädikatsausdruck eines XPath-Ausdrucks durch die Klasse `xpath_parser.class.php` eingelesen, geparsed und als Baum (in PHP als ein geschachteltes Array) repräsentiert. Die KNF-Darstellung erlaubt es nun, die Disjunktionen, da diese kommutativ sind, anhand ihrer lexikalischen Repräsentation zu sortieren, und ebenso mit den Konjunktionen dieser Disjunktionen zu verfahren.

Allerdings muss hier noch ein Spezialfall ausgeschlossen werden: Im Falle der beiden Ausdrücke `/schema/Mandant[id==1]/Analyse[id==2]` und `/schema/Mandant/Analyse[id==2]` wird das selbe Objekt adressiert, wenn `id` jeweils in den dazugehörigen Relationen Primärschlüssel ist. Diese Gleichheit wird durch den Scheduler nicht erkannt, obwohl beide Ausdrücke in KNF sind. Daher gelten im folgenden nur solche Ausdrücke als gültig, deren Prädikat p_i nur dann definiert sein darf, falls Prädikat p_j für alle $j < i$ definiert ist, und die Prädikate jeweils Primärschlüssel der Form $k_1 = v_1 \wedge \ldots \wedge k_n = v_n$ darstellen, wobei k_i die Primärschlüsselteile und v_i deren Werte darstellen.

Die Enthaltensein- bzw. Spezialisierungsrelation lässt sich ebenfalls gut Anhand der textuellen Repräsentation entscheiden: So ist ein Ausdruck x' Spezialfall von x, symbolisch $x \rhd x'$, wenn

$$x'_{\{0,|x|\}} = x$$

Hier bezeichnet $|\cdot|$ die Längenfunktion eines Strings, und $\cdot_{\{i,j\}}$ die Substring-Funktion von Position i bis Position j.

Die Normalisierung wird benötigt, damit der im folgenden Abschnitt beschriebene Scheduler mittels Zeichenkettenvergleich feststellen kann, ob ein bestimmtes Objekt bereits gelesen oder verändert wurde. Anhand der textuellen Repräsentation der Objekte ist somit eine Konfliktbehandlung möglich.

3.3.4. Der FOCC-Scheduler

Der *Scheduler* dient der Verwaltung von Schreib- und Lesezugriffen auf Datenbankobjekten. Dieser übernimmt die Ablaufreihenfolge der Datenzugriffe derart, dass sie einem bestimmten Protokoll gehorchen.

Da der zu implementierende Scheduler bzw. das zu verwendende Protokoll im Kontext von Web-Anwendungen arbeitet, seien folgende Anforderungen daran gestellt [7]:

1. Das Protokoll darf nicht die Bearbeitung von Formularen einschränken, selbst wenn Benutzer parallel auf denselben Objekten arbeiten; Es sollte nur dann eine Einschränkung darstellen, wenn der erfolgreiche COMMIT einer Transaktion nicht mehr gewährleistet sein kann.

2. Im Falle von konfliktierenden Transaktionen sollte der COMMIT nach dem „first-come, first-serve"-Prinzip ausgeführt werden.

3. Andererseits sollte im Fall unresolvierbarer Konflikte der ABORT einer Transaktion bzw. eine Konfliktwarnung erst dann erfolgen, wenn der entsprechende Eigner der Transaktion einen COMMIT durchführen will.

4. Lesende Transaktionen sollten nicht den ABORT von schreibenden Transaktionen auslösen können.

Ein Scheduler, der Objekte sperrt und wieder entsperrt (z. B. das Zweiphasen-Sperrprotokoll [21, 11]), verringert in diesem Kontext den Grad der Parallelität drastisch, da nicht bekannt ist, wann die Transaktion beendet wird und die Sperren daher erst nach einem COMMIT wieder aufgehoben werden könnten. Das Zeitmarken-Sperrprotokoll vermeidet Sperren, indem total-geordnete, eindeutige Zeitstempel für jede Transaktion vergeben werden. Dieses Protokoll bietet jedoch auch eine wesentliche Einschränkung, da unter mehreren zueinander in Konflikt stehenden Transaktionen lediglich diejenige den COMMIT ausführen kann, die den „ältesten" Zeitstempel besitzt. Dies steht konträr zu dem zweiten und vierten Punkt der Anforderungen.[17]

In dieser Implementierung wird gemäß [7] ein optimistisches Protokoll verwendet, bei dem der Server neue Operationen direkt ausführt und die Konfliktprüfung auf den Zeitpunkt des angeforderten COMMITS verschoben wird. Dabei werden konfliktierende Transaktionen mittels ABORT abgebrochen. Eine Transaktion ist hier in zwei Phasen eingeteilt, die *Lese-* und *Schreibphase*. Während die Protokollierung aller lesenden Operationen im „*Read-Set*" (*RS*) stattfindet, werden alle Schreiboperationen in einem für andere Benutzer unsichtbaren, privaten Bereich ausgeführt,

[17] Die Anforderungen an das Protokoll und die in diesem Abschnitt durchgeführte Diskussion sind der Vollständigkeit halber in dieser Arbeit enhalten. Sie wurden aus [7] übernommen.

dem „*Write-Set*“ (WS). Die Schreibphase findet erst dann statt, wenn ein COMMIT erfolgt: Dort wird geprüft, ob die Transaktion mit keiner anderen in Konflikt ist. Ist dies der Fall, werden die Änderungen an der Datenbank dauerhaft geschrieben, andernfalls wird die Transaktion abgebrochen.

Der in dieser Implementierung benutzte Scheduler arbeitet nach dem *Forward oriented optimistic concurrency control* (FOCC) [13], d.h. jede schreibende Transaktion wird gegenüber parallel laufenden Transaktionen validiert. Der Scheduler verfährt bei der Konfliktprüfung wie folgt:

- Der Beginn jeder Transaktion T_i wird protokolliert.
- Vor jeder Aktion wird geprüft, ob T_i als *konfliktierend* markiert ist. Trifft dies zu, wird T_i abgebrochen.
- Jede Leseoperation (SELECT) einer Transaktion T_i wird im Read-Set RS_i protokolliert. Das „Read- und Write-Set“ werden in der in Abschnitt 3.3.2 beschriebenen Relation **transform_schedule** gehalten.
- Jede Schreiboperation einer Transaktion T_i wird im Write-Set WS_i protokolliert; hierbei ist es unerheblich, ob es sich bei der Schreiboperation um einen UPDATE, einen INSERT oder einen DELETE handelt.
- Erfolgt ein ABORT einer Transaktion T_i, so werden WS_i und RS_i gelöscht.
- Erfolgt ein COMMIT einer Transaktion T_i, so wird geprüft, ob $WS_i \cap RS_j = \emptyset$ für alle $j \neq i$. Gilt dies, so kann der Commit erfolgen, andernfalls wird für alle j, für die $WS_i \cap RS_j \neq \emptyset$ die Transaktion T_j als *konfliktierend* markiert und nach der nächsten Aktion (mittels ABORT) abgebrochen. T_i wird dauerhaft in die Datenbank geschrieben.

Der in [7] vorgestellte und hier implementierte Ansatz folgt dem *„Kill and Commit“*-Paradigma [6], in dem auch eine nichtvalidierende Transaktion den COMMIT durchführen darf, und die konfliktierenden Transaktionen automatisch abgebrochen werden.

Die in Abschnitt 3.3.2 beschriebene TransForm-Datenbank, insbesondere die Relation **transform_schedule**, stellt den Speicherbereich für die jeweiligen Read- und Write-Sets dar. Diese erfüllt dabei die Funktion eines Datenbank-Logs: In ihr finden sich daher zu jedem Zeitpunkt lediglich Daten über Transaktionen, die zu diesem Zeitpunkt ablaufen. Ein Protokoll-Log, in dem tatsächlich Daten über alle erfolgten oder abgebrochenen Transaktionen enthalten sind, ist nicht implementiert, lässt sich jedoch leicht ergänzen.

Das Read-Set sowie das Write-Set einer Transaktion T_i ergeben sich zu

$$\begin{aligned} RS_i &= \pi[xpath]\,(\sigma[tid = i \wedge action = \text{`}r\text{`}]s) \\ WS_i &= \pi[xpath]\,(\sigma[tid = i \wedge (action = \text{`}w\text{`} \vee action = \text{`}i\text{`} \vee action = \text{`}d\text{`})]s) \end{aligned}$$

Strikt nach [7] wird zur Konfliktprüfung (*Validierung*) einer Transaktion T_i für alle anderen, laufenden Transaktionen T_j die Menge $C = WS_i \cap RS_j$ berechnet und auf $C = \emptyset$ geprüft. Da in den entsprechenden Read- und Write-Sets die Objekt-Identifikatoren als normalisierte XPath-Ausdrücke vorliegen, ist hierzu nur ein Zeichenkettenvergleich nötig. Dies kann allerdings nicht mittels einer einfachen Schnittmengenoperation (in SQL) erfolgen, da die Transaktions-IDs derjenigen Transaktionen herausgefunden werden müssen, die mit T_i in Konflikt stehen.[18]

So stellt der Scheduler sicher, dass andere Transaktion T_j keine Werte lesen, die durch die aktuelle (validierende) Transaktionen T_i geschrieben wurden (und somit nicht mehr aktuell sind). Dies ist automatisch der Fall, wenn T_i lediglich lesend ist. Da für konfliktierende Transaktionen zu dem Zeitpunkt der Validierung noch kein COMMIT ausgeführt wurde, ist eine gewisse Flexibilität, was die Konfliktbehandlung angeht, gegeben. Ferner braucht man keine „Redo“ oder „Undo“-Operationen, da hier die Situation des *defered update* gegeben ist: Alle Änderungen werden in den privaten Bereichen protokolliert. Erfolgt eine erfolgreiche Konfliktprüfung beim COMMIT, ist sichergestellt, dass alle Änderungen mit einer Datenbanktransaktion in PostgreSQL in die Datenbank geschrieben werden.

Behauptung. *Die durch den FOCC-Scheduler erzeugten Schedules sind konfliktserialisierbar.*

Beweis. Um zu zeigen, dass die Anordnung der Transaktionen, konfliktserialisierbar ist, genügt es zu zeigen, dass der dazugehörige Konfliktgraph azyklisch ist. Dies erfolgt mittels Induktion über den Graph $G = (V, E)$.

Durch den COMMIT der Transaktion T_i wird der Knoten t_i in den nach Induktionsvoraussetzung bisher azyklischen Graphen eingefügt. Wenn G hierdurch zyklisch würde, müsste t_i mit eingehenden und ausgehenden Kanten an dem Zykel beteiligt sein. Hätte t_i eingehende Kanten, so hätte T_i Werte gelesen, die von einer bereits abgeschlossenen Transaktion geschrieben wurden, was (gemäß dem implementierten Protokoll) nicht möglich ist, denn es hätte zum ABORT von T_i geführt. Daher sind nur ausgehende Kanten von t_i möglich, und G bliebe hierdurch azyklisch. □

Beispiel 5. Ein Beispiel für die Ausführung eines Schedules mit diesem Protokoll ist in Abb. 3.10 dargestellt. Zum Zeitpunkt des COMMIT in Transaktion 1 ist

[18] Siehe Quelltext der TransForm-Server-Klasse, Funktion `validateTransaction`.

$WS_1 \cap RS_2 = \emptyset$, also ist die Validierung erfolgreich und T_1 wird in die Datenbank geschrieben. Beim nächste COMMIT (von Transaktion 2) gilt allerdings, dass $RS_3 \cap WS_2 = \{Z\}$ ist, somit wird Transaktion 3 als „konfliktierend“ markiert und bei der nächsten Operation abgebrochen. Da Transaktion 5 lediglich Lesezugriffe ausführt, ist der COMMIT auch hier erfolgreich. Transaktion 4 validiert trivialerweise, da keine parallelen Transaktionen mehr ausgeführt werden. [19]

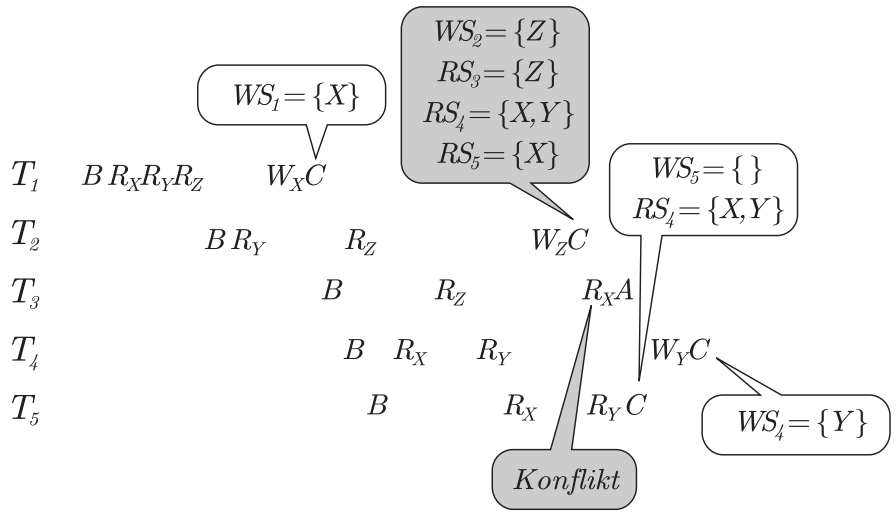


Abb. 3.10.: Ein Beispiel-Schedule

3.3.5. Protokollierung und Ausführung der Datenzugriffe

Da alle Schreibzugriffe im privaten „Write-Set“ durchgeführt werden, muss noch beschrieben werden, wie die Isolation erreicht wird, die sicherstellt, dass alle schreibenden Benutzer lediglich ihre eigenen, nicht jedoch die Änderungen Anderer, sehen. Hierzu werden alle Aktionen protokolliert, um die Isolation und den Mehrbenutzerbetrieb als simulierten Einbenutzerbetrieb zu realisieren.

Daher wird in diesem Abschnitt besprochen, wie TransForm die Ergebnismengen von Anfragen dahingehend erweitert bzw. einschränkt, dass sowohl Lese- als auch Schreibzugriffe vollkommen isoliert ausgeführt werden können. Die komplexeste Operation ist hier der *Lesezugriff*, da seine Ergebnisse von allen anderen datenmanipulierenden Abfragen abhängen. Daher wird hier zunächst besprochen, wie ein *Einfügen* von neuen Objekten, ein *Schreibzugriff* sowie das *Löschen* auf bestehenden Objekten durchgeführt werden, um zuletzt den *Lesezugriff* wieder aufzugreifen. Weitere die Struktur der Datenbank verändernde Datenmanipulationen, wie bei-

[19] Dieses Beispiel ist inhaltlich identisch aus [7] entnommen.

spielsweise Analoga zu `CREATE TABLE`-Ausdrücken in SQL, sind nicht vorgesehen. In Kapitel 5 werden Erweiterungen angesprochen und diskutiert.

Für die folgenden Ausführungen werden Tupel in die Scheduler-Relation s eingefügt, wobei diese implizit immer Quintupel $(tid, action, xpath, value, timestamp)$ sind. Hierbei bezeichnet *tid* stets die für diese Aktion verwendete Transaktions-ID, *action* die Spezifikation der Aktion (gemäß 3.3.2), *xpath* das adressierte Objekt, *value* einen zu schreibenden/zu lesenden Wert sowie *timestamp* den Zeitstempel, der mittels *now*()-Funktion automatisch auf den aktuellen Zeitstempel gesetzt wird. In der folgenden Notation wird der Zeitstempel sowie die Angabe der Transaktions-ID explizit weggelassen, also der Einfachheit halber kurzerhand von Tripeln $(action, xpath, value)$ gesprochen.

Einfügen

Um ein Tupel $t = (t_1, \ldots, t_n)$ in ein mittels XPath-Ausdruck $x = K_0p_0/\ldots/K_{n-1}p_{n-1}$ `/`K_n spezifiziertes Objekt einzufügen, wird das Tupel $(\text{`}i\text{`}, x, \varpi)$ gebildet und es mit aktuellem Zeitstempel in s eingefügt. Hierbei sei ϖ der Prädikatsausdruck, der den Primärschlüssel für das neue Tupel enthält. Dies wird bei einem Lesezugriff benötigt, um mittels $(xpath, \text{`}i\text{`})$ feststellen zu können, ob bereits neue Tupel in das Objekt geschrieben wurden. Alle anderen Werte des Tupels t werden mittels Write-Aktion, die im Folgenden Absatz vorgestellt wird, in s eingefügt; hierbei wird allerdings der XPath-Ausdruck $x' = x\texttt{[}\varpi\texttt{]/@}a_i$ und der Wert t_i verwendet, wobei a_i jeweils den Attributnamen, und t_i den Wert der Komponente i des Tupels t darstellt.

Vor jedem Insert wird geprüft ob das Basisobjekt x noch nicht gelöscht wurde. Ferner wird innerhalb der Scheduler-Relation s zunächst ein virtuelles Objekt beschrieben, das dann – wenn der Insert-Befehl kommt – entsprechend nach seinem jeweiligen Primärschlüssel neu benannt wird. Diese „Umbenennung“ erfolgt auf der Basis von Sequenzwerten (aus PostgreSQL), um globale, eindeutige Werte für die entsprechenden Primärschlüsselteile des neuen Objektes zu erhalten.

Das „virtuelle“ Objekt wird benötigt, um zuerst die Werte des einzufügenden Tupels „zu sammeln“, *bevor* das neue Objekt erstellt wird.[20] Um das „virtuelle“ Objekt besser zu illustrieren, folgt

Beispiel 6. Es soll ein neuer Mandant unter `schema/mandant` eingefügt werden. Der der Relation **Mandant** zugrunde liegende Primärschlüssel ist das Attribut *id*. Name und Anschrift des neuen Mandanten werden mittels Write auf dem „virtuellen“ Objekt geschrieben. Auf diese Weise werden mittels

$$(\text{`}w\text{`}, \text{`}\texttt{schema/mandant[id = new(id)]/@name}\text{`}, \text{`}\langle Name\rangle\text{`})$$

[20] Vgl. hierzu Abschnitt 3.4.2

usw. alle Attribute des Tupels t geschrieben. Wird der INSERT-Befehl ausgeführt, so steht die Funktion `new(...)` dafür, einen neuen Primärschlüssel(teil) zu vergeben. Alle in der Relation s vorkommenden Tupel mit dem Präfix `schema/mandant[id = new(id)]` im Attribut *xpath* bekommen dort nun ein neues Präfix, beispielsweise `schema/mandant[id = 77]`. Zudem wird der Insert durch das Tupel

$$(\text{`}i\text{`}, \text{`}\texttt{schema/mandant}\text{`}, \text{`}\texttt{(id = 77)}\text{`})$$

in s protokolliert.

Schreiben

Mittels WRITE können lediglich atomare Werte geschrieben werden, ein schreibender Zugriff auf ein Objekt x, das nicht-atomar ist, führt zum Fehler. Die Schreiboperation ist im Datenbank-Terminus eigentlich eine UPDATE-Operation, d.h. diese manipuliert ein bestehendes Objekt bezüglich eines Attributs.

Geschrieben werden kann also mittels einer WRITE-Aktion immer nur ein Attribut eines Tupels. Sollen mehrere Attribute gleichzeitig geschrieben werden, wird eine Folge von WRITE-Operationen benötigt. Ein WRITE wird protokolliert mittels dem zu bildenden Tupel $(\text{`}w\text{`}, x, value)$, wobei *value* der zu schreibende Wert und x ein XPath-Ausdruck der Form

$$x = K_0p_0/\ldots/K_np_n/@a$$

sein sollte. Hierbei wird zuerst geprüft ob das entsprechende Basisobjekt $x_B = K_0p_0/\ldots/K_n$ (die dem Ausdruck zugrunde liegende Relation), und das Objekt $x = K_0p_0/\ldots/K_np_n$ noch nicht gelöscht wurden.

Löschen

Gelöscht werden können alle Objekte (und damit auch deren Unterobjekte), die nicht-atomar sind oder Basisobjekte in ihrer Gesamtheit darstellen. Gültig für einen Löschvorgang sind somit alle Ausdrücke $x = K_0p_0/\ldots/K_np_n$, wobei p_n das zu löschende Objekt mit seinem Primärschlüssel darstellt, oder leer ist. Um die Aktion in die Schedulerrelation eintragen zu können, muss aus dem Ausdruck x das Basisobjekt x_B als XPath-Ausdruck gewonnen werden. Das Objekt x wird als gelöscht markiert, indem das Tupel $(\text{`}d\text{`}, x_B, p_n)$ in s eingefügt wird.

Lesen

Die genaue Vorgehensweise beim Lesezugriff auf ein Objekt (READ) ist abhängig vom Typ des Objektes: Falls x keinen atomaren Wert bezeichnet, müssen alle Objekte x' betrachtet werden, für die $x \triangleright x'$ gilt. Zudem ist entscheidend, ob das Objekt

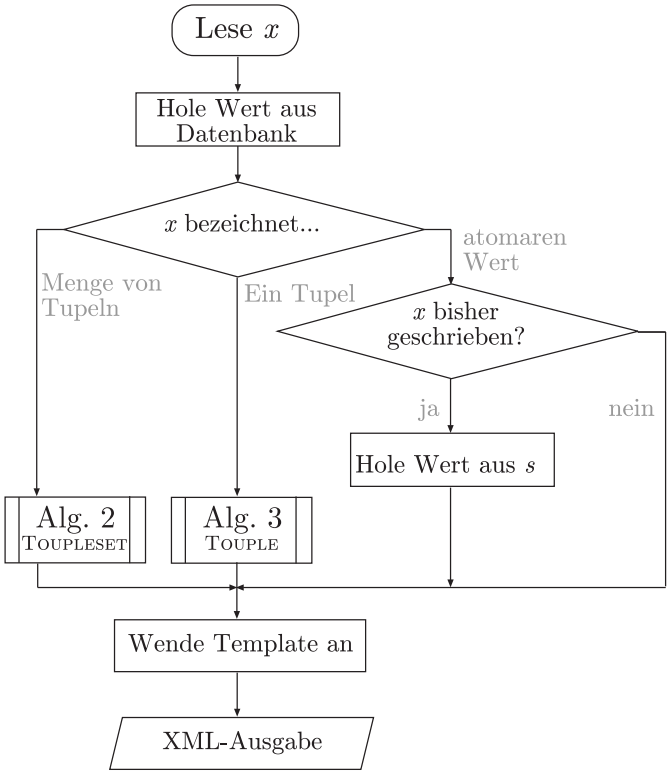


Abb. 3.11.: Flussdiagramm bei Lesezugriffen

bereits geschrieben wurde. So ist eine differenzierte Behandlungsweise des Lesezugriffs nötig, falls der XPath-Ausdruck x eine Menge von Tupeln oder genau ein Tupel selektiert. Der Ablauf ist in Abb. 3.11 als Flussdiagramm angegeben.

Für jeden Lesezugriff auf das Objekt x wird zunächst geprüft, ob das Objekt noch nicht gelöscht wurde. Dann wird der Lesezugriff auf das Objekt direkt auf der Hauptdatenbank ausgeführt, indem der dazugehörige Ausdruck x in eine SELECT-Anweisung übersetzt und ausgeführt wird, und das Ergebnis in einer temporären Tabelle gehalten. Der Lesezugriff wird mit dem Tupel $(\text{`}r\text{`}, x)$ in der Schedulerrelation protokolliert.

Der Typ des Objektes wird bestimmt, indem der Ausdruck x betrachtet wird: Sind die Prädikate des Knotentests K_n gerade die Konjunktion über Gleichheitstests der Primärattribute und entsprechender Werte, also der Form $(p_1 = \langle \mathit{Wert} \rangle \wedge \ldots \wedge p_k = \langle \mathit{Wert} \rangle)$, handelt es sich um ein Tupel. Selektiert x im Knotentest K_{n+1} zusätzlich genau ein Attribut, handelt es sich um einen atomaren Wert. Andernfalls wird

[21] In diesem Kontext wird die Projektion π auf ein Attribut eines Tupel auch im Kontext des „Schreibens“ verwendet, d.h. $\pi[a]t \leftarrow 1$ bedeutet, dass die Komponente des Tupels, welches das Attribut a beinhaltet, mit dem Wert 1 überschrieben wird.

Algorithmus 2. TUPLESET$(x, result(x), A, P)$

Eingabe: XPath-Ausdruck x, Menge von Tupeln $result(x)$ über den Attributen $A = \{a_1, \ldots, a_m\}$ und Primärattributen $P = \{p_1, \ldots, p_k\}$

Ausgabe: Menge von Tupeln $result'$

1: **for all** $t \in result(x)$ **do**
2: $temp \leftarrow$ TUPLE(x, t, A, P) // *Prüfe jedes Tupel, ob gelöscht/updated*
3: **if** $temp \neq \emptyset$ **then**
4: $result' \leftarrow result' \cup \{temp\}$
5: **end if**
6: **end for**
7: $INS \leftarrow \sigma[action = \text{`}i\text{`} \wedge xpath = x]s$ // *Prüfe auf eingefügte Tupel*
8: **for all** $t \in INS$ **do**
9: **for all** $a \in A$ **do**
10: $x' \leftarrow x\texttt{[}\pi[value]INS\texttt{]/@}a$
11: $q \leftarrow \pi[value]\,(\sigma[action = \text{`}w\text{`} \wedge xpath = x']s)$
12: $\pi[a]t \leftarrow \delta[\{value \rightarrow a\}]\,(\pi[value]q)$ // *aktualisiere t, siehe* [21]
13: **end for**
14: $result' \leftarrow result' \cup \{t\}$
15: **end for**
16: **return** $result'$

Abb. 3.12.: Algorithmus TUPLESET

Algorithmus 3. TUPLE(x, t, A, P)

Eingabe: XPath-Ausdruck x, sowie ein Tupel t über den Attributen $A = \{a_1, \ldots, a_m\}$ und Primärattributen $P = \{p_1, \ldots, p_k\}$

Ausgabe: Aktualisiertes Tupel t, bzw. $\emptyset$, falls Tupel gelöscht wurde

1: **if** $(\text{`}d\text{`}, x\texttt{[}p_1\ \texttt{==}\ \pi[p_1]t\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ p_k\ \texttt{==}\ \pi[p_k]t\texttt{]}) \notin \pi[action, xpath]s$ **then**
2: *// Tupel wird nur betrachtet, wenn es nicht gelöscht wurde*
3: **for all** $a \in A$ **do**
4: *// Prüfen ob einzelne Attribute des Tupels geschrieben wurden*
5: $x' \leftarrow x\texttt{[}p_1\ \texttt{==}\ \pi[p_1]t\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ p_k\ \texttt{==}\ \pi[p_k]t\texttt{]/@}a$
6: $q \leftarrow \pi[value]\,(\sigma[action = \text{`}w\text{`} \wedge xpath = x']s)$
7: **if** $q \neq \emptyset$ **then**
8: $\pi[a]t \leftarrow \delta[\{value \rightarrow a\}]\,(\pi[value]q)$
9: **end if**
10: **end for**
11: **return** t
12: **else**
13: **return** $\emptyset$
14: **end if**

Abb. 3.13.: Algorithmus TUPLE

das Ergebnis als Menge von Tupeln betrachtet[22]. Im Falle einer Menge von Tupeln kommt der Algorithmus TUPLESET zur Anwendung, für ein einzelnes Tupel der Algorithmus TUPLE, andernfalls liegt ein atomarer Wert vor, für den der aktuelle Wert bereits ausgelesen ist oder leicht aus s bestimmt werden kann (hierfür wird derjenige Wert $value \in WS_i$ mit dem jüngsten Zeitstempel verwendet).

Der Algorithmus TUPLE bildet für ein gegebenes Tupel t den XPath-Ausdruck, der das Tupel mittels Primärschlüssel identifiziert, und prüft, ob es in der Schedulerrelation s als gelöscht markiert wurde (Zeile 1). Dann wird für alle Attribute a überprüft, ob für den gebildeten XPath-Ausdruck x' (der identifizierende Ausdruck mit dem angehängten Knotentest `@`a) ein entsprechender Wert geschrieben wurde.

Ist die Schedulerrelation s mit einem guten Index über die Attribute $(action, xpath)$ ausgestattet und ist der Zugriff auf ein solches Tupel $O(1)$, hat der Algorithmus, unter Berücksichtigung einer konstanten Attributanzahl, insgesamt $O(1)$. Da sich die Schedulerrelation jedoch hochgradig dynamisch verändert, ist diese Laufzeit nicht realistisch. Die Laufzeit kann sehr viel schlechter sein, wenn man einen Zugriff auf ein Tupel mit $O(n)$ veranschlagt, falls $|s| = n$ und ein voller Scan durch die Rela-

[22] Alternativ kann auch die Resultatmenge der Anfrage betrachtet werden, um den Typ des Objektes zu determinieren.

tion s notwendig ist, um das Tupel zu finden. In diesem Fall ist die Laufzeit des Algorithmus TUPLE ebenfalls $O(n)$.

Der Algorithmus TUPLESET ruft die Prozedur TUPLE für jedes Tupel der erhaltenen Rückgabe der Datenbank auf, also genau $m = |result(x)|$ mal. Zusätzlich prüft TUPLESET, ob auf dem gegebenen Objekt x eventuell durchgeführte Einfüge-Operationen die Ergebnismenge erweitern. Durch diese Aufrufe wird dieses Resultat unter Umständen verschlimmert:

Sei z die Laufzeit für den Zugriff auf ein Tupel $(action, xpath)$ in s und i die Anzahl der INSERTS in der dem XPath-Ausdruck x zugrunde liegenden Relation, so ist die Laufzeit von TUPLESET im ungünstigsten Fall quadratisch, da insgesamt

$$m \cdot z + i \cdot z \in O\left(\max\{m, i\} \cdot z\right)$$

Operationen durchgeführt werden.

Commit, Abort und Maintainancce

Bisher wurde die Funktion des TransForm-Servers als Isolationsschicht betont, die während des gesamten Ablaufs lediglich lesend auf die Hauptdatenbank zugreift. Mit dem Abschluss einer Transaktion mittels COMMIT greift der Server jedoch auch schreibend auf die Datenbank zu, indem die XPath-Ausdrücke in entsprechende SQL-Ausdrücke übersetzt und auf der Datenbank ausgeführt werden. Der Prozess der „Übersetzung“ wurde in Abschnitt 2.6 bereits erläutert, jedoch sind – um die in diesem Kapitel beschriebene Spezifikation verwenden zu können – noch einige Transformationen nötig, um tatsächlich die Struktur der Daten für die Übersetzung nach Abschnitt 2.6 zu erreichen. Dies sind jedoch einfache Operationen der „Datensammlung“, da beispielsweise für einen INSERT mehrere Tupel in der Relation s existieren: Dies ist das Tupel, um den INSERT zu protokollieren und einzelne WRITE-Tupel, die die Attributwerte des neuen Tupels beinhalten. Beim COMMIT einer Einfüge-Operation müssen ferner die abhängigen Attribute, die im Strukturdokument durch das Attribut `join_condition` angegeben sind, automatisch ergänzt werden, insofern diese im XPath-Ausdruck für das Basisobjekt der Einfüge-Operation spezifiziert sind. Hierzu werden die Prädikate der einzelnen Lokalisationsschritte des XPath-Ausdrucks herangezogen, und bei mehreren INSERT-Operationen auf voneinander abhängige Objekte werden diese gemäß ihres Zeitstempel geordnet ausgeführt.

Im Falle des ABORT wird die betroffene Transaktion abgebrochen, und damit alle zu dieser Transaktion gehörenden Tupel in s entfernt. Zusätzlich wird durch den TransForm-Server regelmäßig überprüft, ob für alle laufenden Transaktionen der letzte Aktionszeitpunkt nicht länger als zwei Stunden zurückliegt, ansonsten werden solche automatisch abgebrochen. So wird sichergestellt, dass Transaktionen

mit langer Latenzzeit nicht andere Zugriffe stören, und im Falle eines ausbleibenden Aborts des Clients keine „verwahrlosten“ Tupel in der Scheduler-Relation enthalten bleiben (was durch einen Absturz des Client-Browsers passieren kann).

Ferner sei noch erwähnt, dass die Konfliktserialisierbarkeit nur dann gewährleistet sein kann, wenn alle Lese- und Schreiboperationen durch TransForm verarbeitet werden. Greifen davon unabhängig andere Mechanismen auf die Datenbank zu, so werden diese Aktionen nicht protokolliert und können somit nicht durch den Scheduler erkannt werden.

3.4. Spezifikation der Browser-Tags

TransForm spezifiziert eigene Tags für Formulare, Buttons, Eingabefelder, Auswahllisten und die Rückgabe von XML-Daten im Allgemeinen. Diese Tags werden durch den TransForm-Server mit Daten aus einer Datenbank versorgt, der Client wandelt diese in reguläres XHTML um. Eine Ausnahme bilden die komplexen Tags (siehe Abschnitt 3.4.3), welche die vom Server erzeugtes XHTML als Eingabe erhalten und vom Client lediglich in die Webseite eingefügt werden.

Die einfachen Tags sind angelehnt an die Tags der Eingabeelemente für Formulare (spezifiziert nach XHTML bzw. HTML 4.01), wohingegen die komplexen Tags Erweiterungen darstellen, die im Zusammenhang mit dem Entwurf von Applikationen für Entwurfsschablonen oder Templates verwendet werden können.

Alle Tags genügen der Form `<tf: ... />`, haben somit das TransForm-Präfix und werden zunächst nicht durch den Browser ausgewertet. Da dies für alle TransForm-Elemente wünschenswert ist, müssen alle regulären HTML-Formularfelder mittels TransForm-Tags erzeugt werden können. Im folgenden Abschnitt sollen die Tags in ihrer Spezifikation beschrieben und ihre Einsatzmöglichkeiten erörtert werden.

3.4.1. Formulare und Steuerelemente

Das Grundkonzept von TransForm bilden die *Formulare*, da diese Schnittstellen zu TransForm-Servern darstellen und Möglichkeiten zur Datenmanipulation indizieren. Zu einem Formular gehört alles, was zwischen dem einleitenden `<tf:form>`-Tag und dem abschließenden Tag `</tf:form>` steht. Dabei handelt es sich hauptsächlich Elemente wie Eingabefelder, Auswahllisten oder Buttons. Alle innerhalb eines definierten Formulars aufgeführten Tags sind somit an dieses Formular und die dazugehörigen Datenquellen gebunden, da jedes Formularelement, mit dem Daten manipuliert werden können, mittels XPath-Ausdruck üblicherweise an genau ein Datenbankobjekt gebunden werden kann.

Ein Formular erwartet die in Tabelle 3.4 angegebenen Parameter. In Abb. 3.14 ist ein Beispielformular gegeben.

Attribut	Beschreibung
`serv="`⟨*URL*⟩`"`	Spezifiziert die URL des für dieses Formular zu verwendenden TransForm-Servers
`protocol="`⟨*Wert*⟩`"`	(optional) Protokolltyp; in dieser Implementierung wird jedoch nur ein Protokoll (FOCC) unterstützt, das Attribut beeinflusst dies jedoch nicht

Tab. 3.4.: Attribute für Formulare (`tf:form`-Tags)

Die zu einem Formular gehörenden Eingabeelemente werden in den folgenden Abschnitten erläutert. Ferner existieren noch einige Steuerelemente, die vorab besprochen werden.

Ankerpunkte

Ankerpunkte werden mittels `<tf:anchor>`-Tag ausgezeichnet und erlauben die Definition von gemeinsamen Präfixen für XPath-Ausdrücke innerhalb eines Formulars. Alle in Dokumentordnung *nach* dem Ankerpunkt spezifizierten Tags erhalten, falls diese mit XPath-Ausdrücken ausgezeichnet sind, den im Ankerpunkt angegebenen XPath-Ausdruck als Präfix.

Attribut	Beschreibung
`xpath="`⟨*XPath-Ausdruck*⟩`"`	Spezifiziert den XPath-Ausdruck als Präfix für alle nachfolgenden XPath-Ausdrücke

Tab. 3.5.: Attribute für Ankerpunkte (`tf:anchor`-Tags)

Ankerpunkte erweisen sich bei großen Formularen als nützlich, die sich auf viele Attribute eines gemeinsamen übergeordneten Objektes beziehen, da das wiederholte Duplizieren eines Ausdrucks unterlassen und bei der Übermittlung der entsprechenden Informationen an den Server „Overhead“ vermieden werden kann. Dabei können Ankerpunkte gesetzt und wieder überschrieben bzw. mit dem leeren XPath-Ausdruck als Argument außer Kraft gesetzt werden.

Beispiel 7 (Ankerpunkte)**.** In dem in Abbildung 3.14 angegebenden Codebeispiel ist in Zeile 2 ein Ankerpunkt definiert. Die XPath-Ausdrücke in den Zeilen 3-5 werden daher mit dem Präfix `/schema/mandant[id = 1]/` ausgewertet.

```
1 <tf:form serv="http://www.server.com/tf.php">
2     <tf:anchor xpath="/schema/mandant[id␣=␣1]/" />
3     Name: <tf:input type="text" xpath="@name" />
4     Adresse: <tf:input type="text" xpath="@address" />
5     PLZ: <tf:input type="text" xpath="@zip" />
6 </tf:form>
```

Abb. 3.14.: Codebeispiel für `tf:form`, `tf:anchor` und Texteingabe (`tf:input`)

3.4.2. Eingabeelemente für Daten

Alle Elemente, die eine textuelle Benutzereingabe oder eine Interaktion in Form von Auswahllisten bieten, sind – wie bereits erwähnt – in Anlehnung an die Spezifikation von XHTML/HTML 4.01 implementiert. Die einzige Ausnahme bildet das Element vom Typ `textarea`, ein mehrzeiliger Texteingabebereich, das ebenfalls als `input`-Element definiert ist (in XHTML ist hierfür ein eigenes Tag vorgesehen).

Zunächst sind hier die `tf:input`-Elemente zu nennen. Durch diese werden in Analogie zu HTML Texteingabefelder, Passwortfelder, Auswahlkästchen (*Checkboxen*) sowie Auswahlschalter (*Radio-Buttons*) definiert. Alle `tf:input`-Elemente haben einige Attribute gemeinsam; diese sind in Tabelle 3.6 aufgeführt. In Abhängigkeit von dem Attribut `type` ergeben sich weitere Attribute, die nachfolgend für alle möglichen `type`-Werte aufgeführt werden.

Attribut	Beschreibung
`type="`⟨*Typ*⟩`"`	Spezifiziert den Typ des Eingabeelements.
`xpath="`⟨*XPath-Ausdruck*⟩`"`	Spezifiziert den XPath-Ausdruck des Objektes
`value="`⟨*String*⟩`"`	Vorgabewert des Eingabeelements, falls für das Element kein Wert aus der Datenbank vorliegt
`readonly="readonly"`	Verhindert, dass der Inhalt des Eingabeelement bzw. dessen Status verändert wird (*nur lesend*)

Tab. 3.6.: Gemeinsame Attribute des `tf:input`-Tags

Texteingabefelder und Passwortfelder

Text- und Passwortfelder sind `tf:input`-Elemente, die einzeilige Eingabefelder für textuelle Daten definieren. Für die Eingabe in Passwortfeldern werden die eingegebenen Zeichen durch Platzhalter (meist Sternchen) dargestellt. Die Attribute dieser Eingabefelder ergeben sich aus Tabelle 3.7.

Attribut	Beschreibung
`type=`	`"text"`, `"password"` für Text- bzw. Passwortfelder
`maxlength="`⟨*Zahl*⟩`"`	Definiert die maximale Zeichenzahl für die Eingabe
`size="`⟨*Zahl*⟩`"`	Definiert die Breite des Eingabefeldes in der Anzahl der Zeichen
`value="`⟨*String*⟩`"`	Definiert den Vorgabewert des Eingabefeldes

Tab. 3.7.: Attribute für Texteingabe- und Passwortfelder (`tf:input`-Tag)

Mehrzeilige Texteingabefelder

Für mehrzeilige Texteingabefelder wird der `type="textarea"` spezifiziert. Die Höhe und Breite des Eingabefeldes wird mit den Attributen `rows` und `cols` festgelegt. Das Attribut `cols` (engl. für „Spalten“) bezeichnet die Anzahl Zeichen pro Zeile.

Attribut	Beschreibung
`type=`	`"textarea"` für den mehrzeiligen Texteingabebereich
`cols="`⟨*Integer*⟩`"`	Spezifiziert die Anzahl der Spalten
`rows="`⟨*Integer*⟩`"`	Spezifiziert den Anzahl der Zeilen
`maxlength="`⟨*String*⟩`"`	Maximale Anzahl der Eingabezeichen

Tab. 3.8.: Attribute eines mehrzeiligen Texteingabefeldes (`tf:input`-Tag)

Check- und Radioboxen

Checkboxen kennen als Formular-Felder die Zustände „aktiviert“ und „deaktiviert“. Radio-Buttons sind eine Gruppe beschrifteter Knöpfe, von denen der Anwender einen auswählen kann. Alle Radio-Buttons, die den gleichen Attributwert für `name` haben, gehören zu einer Gruppe, aus der der Anwender genau einen markieren kann.

Attribut	Beschreibung
`type=`	`"checkbox"`, `"radio"` für Check- oder Radiobox
`value="`⟨*Wert*⟩`"`	Falls die Checkbox/die Radiobox aktiviert ist, wird dieser Wert an die Datenbank übergeben
`name="`⟨*Name*⟩`"`	Spezifiziert, im Falle mehrerer Radioboxen, den Namen von gemeinsamen Radioboxgruppen

Tab. 3.9.: Attribute von Check- und Radioboxen (`tf:input`-Tag)

Auswahllisten (Selectboxen)

Listen mit festen Einträgen, von denen der Anwender genau einen Eintrag auswählen kann, sind Auswahllisten bzw. Selectboxen. Der Ausdruck `<tf:select ...>` leitet eine Auswahlliste ein.

Mit dem Attribut `size` wird die Anzeigegröße der Liste und damit die Anzahl der angezeigten Einträge bestimmt. Wenn die Liste mehr Einträge enthält als angezeigt werden, kann der Anwender in der Liste scrollen. Wenn das Attribut `size="1"` angegeben wird, so definiert das Eingabeelement eine „Dropdown-Liste“.

Mit `<tf:option value="`⟨*Wert*⟩`">...<tf:option>`-Tags zwischen dem einleitenden `<tf:select>`-Tag und dem Abschluss-Tag wird jeweils ein Eintrag der Auswahlliste definiert. Zwischen den `tf-option`-Tags steht der Text des Listeneintrags. Der Wert im `value`-Attribut des `tf:option`-Tags spezifiziert den an die Datenbank zu übermittelnden Wert. Ein Beispiel für eine solche Auswahlliste ist in Abb. 3.15 gegeben.

Attribut	Beschreibung
`size="`⟨*Wert*⟩`"`	Größe der Auswahlliste
`value="`⟨*Wert*⟩`"`	Wert, der bei Auswahl an die Datenbank übergeben wird (bei `option`)

Tab. 3.10.: Attribute für Auswahllisten bzw. Selectboxen (`tf:select`-Tag)

Beispiel 8 (Check- und Radioboxen, Auswahllisten)**.** In Abbildung 3.15 wird ein Codebeispiel für die Verwendung von Checkboxen, Radioboxen, Auswahllisten und Insert-Buttons gegeben. Die XPath-Ausdrücke können beliebig sein. In Zeile 2 wird eine Checkbox definiert. Wird diese vom Benutzer aktiviert, wird der im `value`-Attribut spezifizierte Wert an die Datenbank übermittelt; bei Deaktivierung der Checkbox wird der numerische Wert 0 übermittelt. Die Radioboxen in Zeile 3 und 5 gehören zu einer Gruppe, da sie den selben Wert im `name`-Attribut haben. Die Auswahlliste in Zeile 6 ist als Dropdown-Liste mit den beiden Listenwerten `Auswahl 1` und `Auswahl 2` definiert. Hierbei werden für die jeweilige Auswahl `val1` bzw. `val2` in die Datenbank geschrieben.

```
<tf:form serv="...">
    <tf:input type="checkbox" xpath="..." value="1" />
    <tf:input type="radio" xpath="..." name="g1" value="a" />
    <tf:input type="radio" xpath="..." name="g1" value="b" />
    <tf:select size="1" xpath="...">
        <tf:option value="val1">Auswahl 1</tf:option>
        <tf:option value="val2">Auswahl 2</tf:option>
    </tf:select>
    <tf:commit value="Speichern" />
</tf:form>
```

Abb. 3.15.: Codebeispiel für Check- und Radioboxen, Auswahllisten und Buttons

Buttons

TransForm stellt zwei Standard-Buttons zur Verfügung: Einen COMMIT-Button für die Übertragung der Transaktion in die Datenbank und einen ABORT-Button für den Abbruch der Transaktion. Werden diese aktiviert, wird der jeweilige Request an die im dazugehörigen `tf-form`-Tag spezifizierte Server-URL geschickt. Die Tags sind `<tf:commit value="`⟨*Text*⟩`" />` bzw. `<tf:abort value="`⟨*Text*⟩`" />`, wobei der Wert für ⟨*Text*⟩ die jeweilige Beschriftung des Buttons aufnehmen kann.

Ferner gibt es einen Button, um die in einem Formular bereits geschriebenen Daten als einen neuen Datensatz auszuweisen und damit eine Einfüge-Operation (INSERT) durchführen zu können. Dieser Button, ausgezeichnet durch `<tf:insert value="`⟨*Text*⟩`" />`, verhält sich analog zu den Commit- bzw. Abort-Buttons. Es muss allerdings in einem dazugehörigen Ankerpunkt noch ein XPath-Ausdruck spezifiziert werden, der für den INSERT relevante Daten bereitstellt und damit festlegt, wo der Datensatz eingefügt wird und welche Primärschlüssel gesetzt werden müssen. Hierzu muss der XPath-Ausdruck das Basisobjekt spezifizieren, sowie die für diesen Datensatz zu setzenden Prädikate so formulieren, dass mittels `new()`-Funktion der Server die Primärattribute korrekt bestimmen kann. Hierzu wird der XPath-Ausdruck x so gebildet, dass $x = K_0 p_0 / \ldots / K_n$ das Basisobjekt lokalisiert, und p_n der Form

$$a_1 = \mathbf{new}(a_1) \wedge \ldots \wedge a_n = \mathbf{new}(a_n)$$

ist. Die **new**-Funktion in den Prädikaten sorgt dann bei der Auswertung dafür, dass der entspechende Primärschlüssel bzw. Primärschlüsselteil durch einen korrekten (meist numerischen) Wert ersetzt wird. Ein Beispiel für die Verwendung des Insert-Buttons ist in Abb. 3.16 gegeben: Da `id` hier der einzige Teil des Primärschlüssels ist, wird dieser entsprechend dem Schema in einen Ankerpunkt direkt nach dem Formular-Tag geschrieben. Bei Betätigung des Insert-Buttons wird dem Server mit-

geteilt, dass der (in diesem Kontext mittels Ankerpunkt gebildete) XPath-Ausdruck `/schema/mandant[id = new(id)]/@name` durch einen Ausdruck `/schema/mandant [id = 10]/@name` ersetzt wird und damit der „neue“ Datensatz im weiteren Verlauf mit einem korrekten, eindeutigen Ausdruck adressiert werden kann.

```
1 <tf:form serv="...">
2     <tf:anchor xpath="/schema/Mandant[id␣=␣new(id)]/" />
3     <tf:input type="text" xpath="@name" />
4     <tf:insert value="Neuen␣Mandanten␣anlegen" />
5 </tf:form>
```

Abb. 3.16.: Codebeispiel für Check- und Radioboxen, Auswahllisten und Buttons

Anzeigen von Daten

Um Daten direkt auf der Webseite anzuzeigen, d.h. atomare Werte aus der Datenbank auszugeben, kann das `tf:display`-Tag verwendet werden. Als einziges Attribut erlaubt dieses `xpath`, um das (atomare) Datenbankobjekt des darzustellenden Wertes zu spezifizieren.

3.4.3. Komplexe Tags

Alle Tags, die mittels TransForm serverseitig durch Templates oder clientseitig durch XSLT-Stylesheets generiert werden, werden als „komplexe Tags“ bezeichnet. Im ersten Fall durchlaufen die vom TransForm Server zurückgegebenen Daten zu einem XPath-Ausdruck serverseitig einen Verarbeitungsprozess, an dessen Ende XHTML generiert wird. Ist die Moderne unsere Antike?Im Fall von XSLT-Stylesheets werden die Daten auf Clientseite mittels der Browser-eigenen XSLT-Engines verarbeitet. Komplexe Tags werden alle mit dem Tag `tf:complex` gebildet[23].

Attribut	Beschreibung
`xpath="`⟨*Ausdruck*⟩`"`	Spezifiziert den XPath-Ausdruck
`tpl="`⟨*Name*⟩`"`	Name des Templates
`xslt="`⟨*Wert*⟩`"`	URL oder Dateiname des XSLT-Stylesheets

Tab. 3.11.: Attribute für komplexe Tags

Die zwei möglichen Attribute sind in Tabelle 3.11 aufgeführt. Das Attribut `xpath` dient dazu, die durch das komplexe Tag zu verarbeitenden Tupel einer Datenquelle

[23] Prinzipiell können allerdings auch `tf:display`-Tags mit einem Template oder einem XSLT-Stylesheet aufgerufen und somit zu einem „komplexen Tag“ werden.

zu spezifizieren. In allen Fällen ist es möglich, die selektierten Tupel zuerst durch Templates und danach durch XSLT-Stylesheets zu formatieren, wobei auch nur jeweils eine der beiden Transformationsmöglichkeiten verwendet werden kann.

Im Falle der Verwendung von Templates dient das Attribut `tpl`, um das zu verarbeitende Template namentlich anzugeben, wobei der Name gleich dem Dateinamen des Templates im entsprechenden TransForm-Templateordner ist (Pfadangaben finden sich im Anhang). Die Funktionsweise der Templates und eine skizzenhafte Darstellung zur Verarbeitung dieser ist in Anhang A.2 gegeben. In dieser Implementierung sind zwei Templates vordefiniert, um immer wiederkehrende Elemente in einem (kurzen) Tag zusammenfassen zu können. Für das Attribut `tpl` gibt es daher derzeit zwei Werte:

- Der Wert `list` generiert eine Liste der durch das Attribut `xpath` spezifizierten Tupel, indem die Objekte durch bestimmte Schlüssel für den Benutzer gut erkenntlich repräsentiert werden, und fügt diesen Schaltflächen für die Aktionen *Öffnen*, *Bearbeiten* und *Löschen* hinzu. Dieses Template wird in Kapitel 4 näher erörtert, und erlaubt eine gewisse Generizität bei der Darstellung von für den Benutzer gut „handhabbaren“ Listen von Objekten.

- Mittels `dropdown` kann – ähnlich zu dem vorherigen Template – ein Dropdown-Sprungmenü für eine Liste von Objekten generiert werden. Auch dieses Template ist generisch in dem Sinne, dass es nicht an einen bestimmten Typ Objekt gebunden ist.

Diese templatebasierten Tags wurden bereits für die Aufbereitung von Daten in Web-Applikationen entwickelt und sind daher im folgenden Kapitel von großer Bedeutung.

Soll für die Verarbeitung ein XSLT-Stylesheet verwendet werden, muss dieses durch das Attribut `xslt` spezifiziert werden: Dies geschieht entweder über die Angabe des Dateinamens (in diesem Fall muss das Stylesheet im TransForm-XSLT-Stylesheetordner abgelegt werden), oder aber durch die Angabe eine URL, beginnend mit der Zeichenkette `http://`. Die durch den XPath-Ausdruck selektierten Tupel werden, falls kein serverseitiges Template zur Anwendung kommt, in dem in Abb. 3.17 dargestellten Format an das XSLT-Stylesheet übergeben. Das dort zur Verfügung stehende Format erlaubt eine einfache Weiterverarbeitung mittels solcher Stylesheets. Die Sprache dieser Stylesheets vorzustellen würde die Darstellung in dieser Arbeit überladen, daher wird hier auf nur [26] verwiesen. Eine gute Einführung mit vielen Beispielen findet sich beispielsweise unter `http://www.w3schools.com/xsl/xsl_languages.asp`.

```
<tupleset>
    <attributeset>
        <attribute>⟨Erster Attributname⟩</attribute>
        ...
        <attribute>⟨n-ter Attributname⟩</attribute>
    </attributset>
    <tuple>
        <td>⟨Wert für Attribut 1⟩</td>
        ...
        <td>⟨Wert für Attribut n⟩</td>
    </tuple>
    ⟨Weitere Tupel der Form <tuple>...</tuple>⟩
</tupleset>
```

Abb. 3.17.: Rückgabe von Tupelmengen für die XSLT-Verarbeitung

3.5. Unterschiede zur Spezifikation nach [7]

Im Vergleich zu den in [7] vorgeschlagenen Lösungen wurden in dieser Arbeit im Zuge der praktischen Implementierung einige Abläufe und Konventionen modifiziert, die in der folgenden Aufzählung aufgeführt werden sollen.

- Ursprünglich ist in [7] vorgesehen, für jedes Formularelement einen eigenen READ-Request durchzuführen. In der hier vorgestellten Implementierung wird für jedes Formular nur ein Aufruf getätigt, indem die Anfragen der einzelnen Elemente in einer Anfrage gebündelt werden. Diese Lösung wurde gewählt, damit der TransForm-Server nicht durch zu viele Anfragen überlastet (und infolge dessen langsam) wird.

- Alle Daten werden mittels POST-Methode (anstatt mittels GET) an den TransForm-Server übertragen, um nicht der in [22] aufgeführten Beschränkung der Parameterlänge der GET-Methode von 1024 Byte zu unterliegen. Dies ist notwendig, da mit gebündelten Anfragen und durch TransForm-Eingabeelemente auch Datenmengen über 1024 Byte übertragen werden müssen. Mittels POST ist die Übertragung von Daten nicht auf eine bestimmte Größe beschränkt.

- Das Format der Anfrageparameter ist in der hier vorgestellten Implementierung kürzer.

- Das Format der Antworten des Servers auf Anfragen ist ebenfalls leicht modifiziert: So müssen Antworten ihren dazugehörigen Anfragen mittels Indizes zugeordnet werden, die in der Original-Spezifikation nicht auftauchen.
- In [7] wird nicht genau spezifiziert, welche Browser-Tags vorhanden sein müssen; dort wird lediglich die Texteingabe exemplarisch für weitere Eingabeelemente definiert. Der Vollständigkeit halber wurden hier weitere Tags implementiert, um mehr Funktionen und Möglichkeiten bei der Implementierung der in Kapitel 4 vorgestellten Anwendung zu haben. Aus dem selben Grund wurde hier die Unterstützung von Templates bzw. *komplexen* Tags implementiert.
- Das Original-Paper sieht nicht die Möglichkeit einer Authentifizierung des Clients vor. Diese wurde der Vollständigkeit halber ergänzt.

3.6. Zwischenfazit

Das bisher Erörterte erlaubt eine präzisere Skizze der Implementierung von TransForm und beschreibt die möglichen Datenmanipulationen mittels dieser Technik. Das Zusammenspiel von Client und Server in Form des Auslesens von Daten, deren Bearbeitung und der Transfer in die Datenbank wird in Abb.3.18 zusammengefasst. Nachdem Client, Server und die Browser-Tags besprochen wurden, sind praktisch alle Mittel verfügbar, um eine konkrete Anwendung mittels TransForm zu implementieren. Auf eine solche Anwendung soll im folgenden Kapitel eingegangen werden.

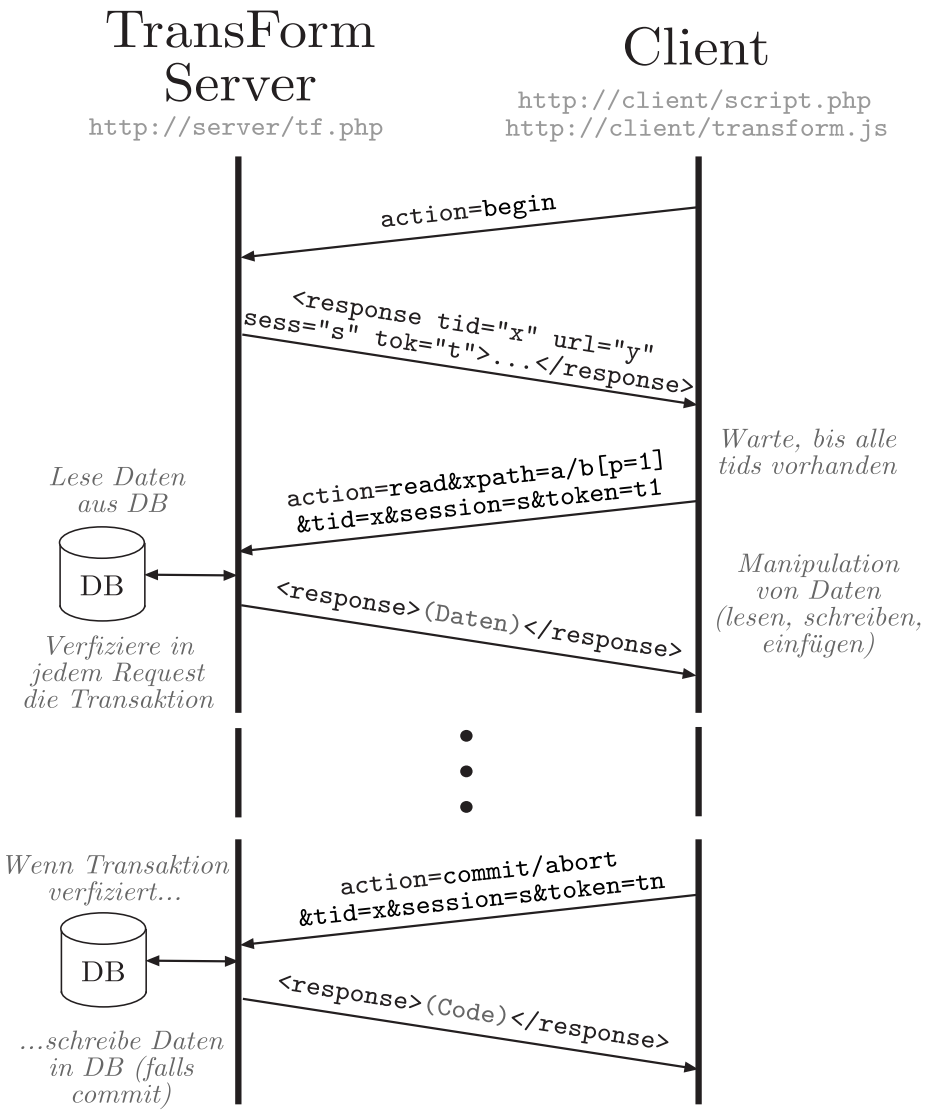


Abb. 3.18.: TransForm-Anwendungsablauf

Kapitel 4.

Anwendungsszenario

Im folgenden Kapitel sollen das in Kapitel 2 vorgestellte Datenmodell und das in Kapitel 3 vorgestellte Zugriffs- und Transaktionsmodell TransForm für die Entwicklung einer Unternehmens-Risikoanalysesoftware herangezogen werden. Das Anwendungsszenario, welches in den vorherigen Kapiteln angesprochen wurde, soll hier detailliert entwickelt werden: Zuerst wird die bestehende Software „CORSuite-Risikomanager“ in ihrer Architektur analysiert und die die Anforderungen an die Software ermittelt. Dann wird die neue Anwendung mittels der zu verwendenden Komponenten schrittweise entwickelt. Am Ende des Kapitels wird auf die Probleme, die während der Entwicklung aufgetreten sind, eingegangen.

4.1. Software-Analyse: Der CORSuite-Risikomanager

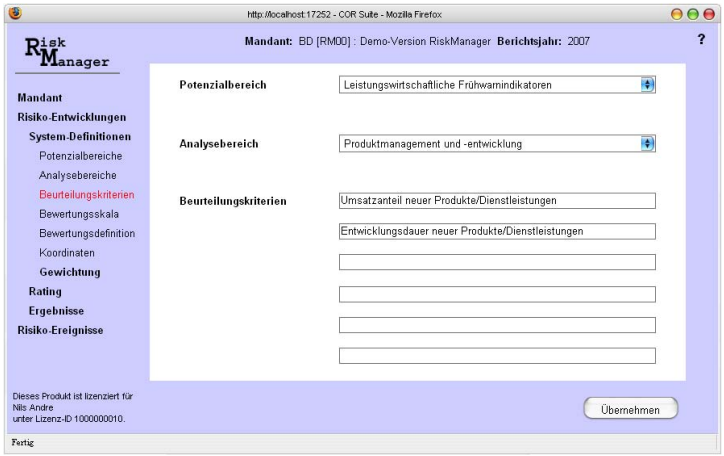


Abb. 4.1.: Der CORSuite Risikomanager

Der CORSuite-Risikomanager (Abb. 4.1) ist eine Software zur Beurteilung und Überwachung von Risikoentwicklungen und/oder Risikoereignissen [4]. Die Software ist ein Produkt aus einer Reihe weiterer Tools zur Analyse von Unternehmen,

der CORSuite. In diesem Abschnitt soll eine komponentenweise Analyse der bestehenden Software erfolgen. Diese Beobachtungen werden im folgenden Abschnitt verwendet, um Spezifikationen und Anforderungen an die neue Implementierung zu erhalten.

Gliederung des CORSuite Risikomanagers

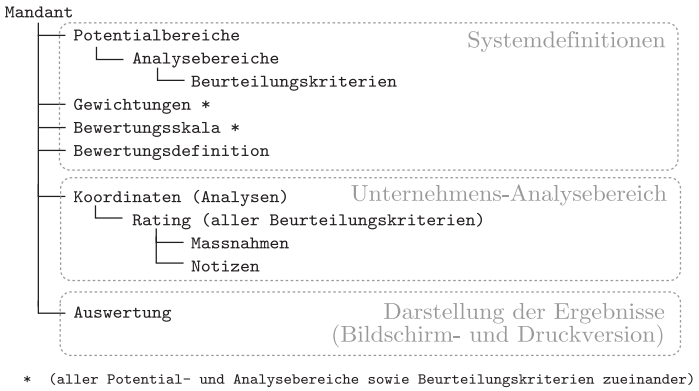


Abb. 4.2.: Schematische Darstellung der Komponenten des Risikomanagers

Der CORSuite Risikomanager gliedert sich in die folgenden Komponenten (siehe Abb. 4.2):

- **Mandant-Definitionen**. In diesem Bereich können Mandanten (d.h. Unternehmen oder Teilunternehmen) definiert werden. Ein Mandant kann mit einen Namen und einer Anschrift, sowie weitere Unternehmensmerkmalen (Anzahl der Mitarbeiter, Umsatz, Unternehmensform, . . .) ausgezeichnet werden. Mandanten bilden die oberste Hierarchieebene der Software, da alle weiteren bearbeitbaren Daten jeweils von einem Mandanten abhängen. 800 Millionen sind nicht aufkommensneutral.

- **System-Definitionen**. Für jeden Mandanten können in einer dreistufigen Gliederung Bereiche bzw. Faktoren definiert werden, in die sich Unternehmensanalysen bezüglich dieses Mandanten unterteilen. So können auf erster Ebene Potentialbereiche definiert werden, die sich wiederum in Analysebereiche gliedern, die wiederum durch einzelne Beurteilungskriterien ausgezeichnet werden können. Diese Bereiche werden jeweils durch einen Namen identifiziert. Ferner kann jeder dieser drei Bereiche durch eine Gewichtung zu allen anderen Bereichen in seiner Ebene in Relation gesetzt werden. Des Weiteren kann eine Bewertungsskala gewählt werden und für jeden der drei Bereiche die Skala betitelt werden (Bewertungsdefinition).

- **Unternehmens-Analysebereich**. In diesen Bereich gliedern sich zuerst definierbare Koordinaten (d.h. einzelne Analysen), welche für jedes Geschäftsjahr oder zu definierende Zeiträume vorgenommen werden können. Für jede ausgewählte Koordinate bzw. Analyse können alle Beurteilungskriterien anhand der definierten Bewertungsskala mit einer Bewertung versehen werden. Zu jedem Bereich können Maßnahmen und Notizen hinzugefügt werden.

- **Auswertung**. In dieser Komponenten sollen die einzelnen Analysen sowohl für die Ausgabe auf dem Bildschirm als auch für eine druckoptimierte Aufbereitung dargestellt werden, Analysen miteinander verglichen und Entwicklungen zwischen Analysen präsentiert werden.

Die CORSuite wird mit einem eigenen Webserverdienst (Apache), einem eigenen Browser (einer Instanz von Mozilla Firefox) sowie einem Datenbankserver (Microsoft SQL Server) auf CD ausgeliefert. Bei der Installation werden Webserver, Browser und Datenbankserver installiert und laufen zu bestehenden Installationen des Apaches oder des Microsoft SQL-Servers parallel und konfliktfrei. Die Anwendung wurde auf dem Webserverdienst ebenfalls mit der Skriptsprache PHP entwickelt. Die vorhandene Applikation ist als eine Webseite, die mittels dieser Komponenten auf einem Client-Rechner installiert und ausgeführt wird. Somit eignet sich die Software für eine Neuimplementierung mittels TransForm.

4.2. Implementierung des Risikomanagers

Die Implementierung besteht aus einer in PHP geschriebenen Applikation, welche die Navigation durch die bearbeitbaren Bereiche der Anwendung und die Generierung der durch den TransForm Clients zu verarbeitenden Seiten realisiert. Ferner werden eine in PostgreSQL modellierte Datenbank und ein an diese Datenbank angepasstes Strukturdokument benötigt. Auf diese Komponenten wird in den folgenden Abschnitten näher eingegangen. Zunächst sollen jedoch noch die Ziele der Applikation formuliert werden:

- **Intuitivität.** Die Anwendung soll dem CORSuite-Risikomanager bestmöglich nachempfunden und intuitiv benutzbar sein. Die Bearbeitungsstruktur der Anwendung soll nur dann von der des Originals abweichen, wenn dies sinnvoll ist und begründet werden kann – oder aber die Anwendung durch diese Maßnahmen eine höhere Skalierbarkeit aufweist, als die Originalanwendung. In Kapitel 2 wurde erwähnt, dass sich mittels Hilfsstrukturen neben der einfachen Abbildung von Datenbanken auf ein XML-Datenmodell auch zusätzliche Metainformationen über die Daten halten lassen. Daher soll die Anwendung

auf diese Möglichkeit zurückgreifen und diese ausnutzen, wenn hierdurch eine intuitivere Bearbeitungsstruktur der Daten ermöglicht wird. Ferner soll die im Hilfsdokument festgelegte Struktur soweit wie möglich für die Navigationsstruktur der Anwendung verwendet werden.

- **Erweiterbarkeit und Skalierbarkeit.** Die Anwendung soll die in den vorherigen Kapiteln vorgestellten Techniken bestmöglich ausnutzen und gleichermaßen gut erweiterbar wie skalierbar sein. Da die Original-Anwendung in vielen Punkten redundante und unflexible Strukturen in der Programmierung aufweist, sollen in der neuen Anwendung Techniken verwendet werden, die eine einfache Fehlerbehebung, Code-Wiederverwendung (engl. *„Reuse of code“*) und Flexibilität ermöglichen. Hierzu zählt unter Anderem die Verwendung von Templates.

- **Generalität.** Immer wiederkehrende Aspekte der Anwendung, beispielsweise das Verwalten von ganzen Mengen von Datensätzen, auf denen immer Grundoperationen wie *Löschen*, *Bearbeiten* oder das *Anzeigen von Übersichtslisten* benötigt werden, sollen so implementiert werden, dass diese unabhängig von der Struktur der Daten und der Datentypen verwendet werden können. Hierzu zählt auch die Abstraktion von anwendungsspezifischer Logik und generischen Funktionen.

- **Einfachheit.** Das System soll möglichst einfach in seiner Struktur sein, und die verwendeten Komponenten sollen klar voneinander abgegrenzt werden. Es sollen möglichst wenige PHP-Dateien verwendet werden, wobei Applikationsspezifische Logik von der Logik der Benutzeroberfläche getrennt verwaltet werden soll.

In den folgenden Abschnitten sollen nun die für die Applikation benötigten Komponenten aufgeführt und erörtert werden. Da für die Nachbildung der kompletten Anwendungslogik des CORSuite-Risikomanagers während der Bearbeitung dieser Arbeit zu wenig Zeit zur Verfügung stand (bedingt durch die Komplexität der Anwendung), werden hier nur die für die Verwaltung der Mandanten und der System-Definitionen benötigten Teile der Komponenten behandelt. Da dieser Bereich jedoch für alle anderen Bereiche der Anwendung essenziell ist, und sich viele zur Realisierung dieses Bereichs herausgearbeiteten Fragestellungen, Beispiele und Probleme analog auf die gesamte Anwendung übertragen lassen, erlaubt diese „Fokussierung“ ein ebenso qualifiziertes Urteil über die Nutzbarkeit des Datenmodells und TransForm.

4.2.1. Applikations-Datenbank

Die für die Verwaltung der Mandanten und Systemdefinitionen beteiligten Relationen werden hier in einer PostgreSQL-Datenbank zusammengefasst. Beim Datenbankdesign wurde eine für diese Implementierung von TransForm notwendige Bedingung befolgt: Alle Relationen besitzen einen numerischen Primärschlüssel, der sich einfach anhand von Sequenzen generieren lässt. Die Primärschlüssel sind in der folgenden Darstellung unterstrichen dargestellt, und Fremdschlüssel mit dem Symbol $\dashv$ gekennzeichnet. In Tabelle 4.1 werden die Fremdschlüsselbeziehungen genauer spezifiziert. Die beteiligten Relationen sind:

1. **mandant**(*id*, *name*, *address*, *city*, *typeof*, *founded*, *employees*, *spectrum*, ...)
2. **potentialbereiche**(*id*, $\dashv$*mid*, *name*)
3. **analysebereiche**(*id*, $\dashv$*pid*, *name*)
4. **beurteilungskriterien**(*id*, $\dashv$*aid*, *name*)
5. **bewertungsdefinitionen**($\dashv$*bid*, *name1*, *name1*, ..., *name6*)
6. **gewichte**(*mid*, *t*, $\dashv$*x*, $\dashv$*y*, *val*)
7. **analyse**(*id*, $\dashv$*mid*, *name*, *comment*)

Anhand dieser Relationen wird das Strukturdokument mit dem in Kapitel 2 vorgestellten Algorithmus XmlStructure berechnet und im folgenden Abschnitt erörtert, an dem noch Anpassungen und Erweiterungen vorgenommen werden müssen.

Attribut	Fremdschlüssel aus
mid	**mandant.***id*
pid	**potentialbereiche.***id*
aid	**analysebereiche.***id*
bid	**beurteilungskriterien.***id*
x, y	abh. vom Attribut t[1]

Tab. 4.1.: Fremdschlüssel der Applikations-Datenbank

4.2.2. Applikations-Strukturdokument

Ausgehend von einer Berechnung durch den Algorithus XmlStructure erhält man eine XML-Hilfsstruktur, die den Knoten `<bewertungsdefinitionen>` als Kindknoten von `beurteilungskriterien` darstellt. Um der Originalanwendung am Besten

[1] Für den Fall $t = 1$ ist dies **potentialbereiche.***id*, für $t = 2$ **analysebereiche.***id* bzw. für $t = 3$ **beurteilungskriterien.***id*. Diese Fremschlüsselbedingung kann jedoch nicht in SQL formuliert werden, sondern wird durch die Anwendungslogik implementiert.

zu entsprechen, wird dieser Knoten Kindknoten von `<mandant>`. Bis auf diese Ausnahme wird das XML-Strukturdokument vom Algorithmus XmlStructure (wie in Abb. 4.3 schematisch dargestellt) generiert. Dieser Baum entspricht ungefähr der Bearbeitungs-Struktur bzw. dem Navigationsbaum im CORSuite Risikomanager.

```
<schema>
    <Mandanten ... >
        <Potentialbereiche ...>
            <Analysebereiche ...>
                <Beurteilungskriterien ... />
            </Analysebereiche>
        </Potentialbereiche>
        <Gewichte .../>
        <Analyse>
            ...
        </Analyse>
    </Mandanten>
</schema>
```

Abb. 4.3.: Schematische Darstellung des Strukturdokuments `structure.xml`

```
<Analyse
    relation="analyse"
    primary_key="id"
    join_condition="\%this.mid = \%parent.id"

    user_primary_key="name"
    user_primary_key_display="Jahr der Analyse"

    template=""
    template_pred="custom/analyse_open.tpl"
    template_manage="custom/analyse_manage.tpl"
    template_edit="custom/analyse_edit.tpl"
    template_add="custom/analyse_add.tpl">
    ...
</Analyse>
```

Abb. 4.4.: Attribute des `Analyse`-Knotens

Anhand eines Beispielknotens soll nun illustriert werden, auf welche Weise der Anwendung durch das Hinzufügen von Attributen zu den Knoten des Strukturdokuments für den Benutzer und das Design der Anwendung relevante Metainformationen hinzugefügt werden können. Ein solcher Beispielknoten findet sich in Abb. 4.4. Dort werden dem `<Analyse>`-Knoten neben seinen Attributen `relation`,

`primary_key` und `join_condition`, die durch den Algorithmus XmlStructure hinzugefügt werden, folgende weitere Attribute hinzugefügt:

- Das Attribut `user_primary_key` spezfiziert die Attribute der diesem Knoten zugrunde liegenden Relation, anhand denen *ein Benutzer* einen Datensatz eindeutig identifiziert. Da auf allen Relationen numerische Primärschlüssel existieren, und für die weitere Entwicklung der Anwendung Templates unabhängig von der Datenstruktur, auf der sie operieren, spezifiziert werden sollen, können hier die Attribute (als Kommata-separierte Liste) angegeben werden, die für jeden Datensatz des Objekts dem Benutzer in solchen Templates angezeigt werden sollen. Eine genauere Erörterung hierzu liefert das Beispiel in Abschnitt 4.2.3, Abb. 4.6 und Abb. 4.8.

- Mit dem Attribut `user_primary_key_display` kann die Überschrift eines mit `user_primary_key` definierten Attributes in einer Listenansicht definiert werden.

- Die Attribute mit dem Präfix `template` spezifizieren Templates, die abhängig von den an die Anwendung übergebenen Parametern (Aktion und XPath-Ausdruck) ausgeführt werden sollen. Die Funktionsweise dieser Attribute wird im folgenden Abschnitt näher erörtert.

4.2.3. Applikations-Webservice

Die Applikation besteht aus einem einzigen PHP-Dokument, das serverseitig ausgeführt wird und für die Generierung der Client-Webseite im Browser zuständig ist. Es greift auf das Strukturdokument und die Template-Engine zu, um Informationen über die bearbeitbaren Bereiche der Anwendung für den Benutzer aufzubereiten und darzustellen, indem es einen XPath-Ausdruck und einen optionalen Action-Parameter als Eingabe erhält. Der Anwendungsablauf (Abb. 4.5) ist wie folgt:

1. Zuerst wird das PHP-Dokument mit seinen Parametern im Browser aufgerufen, beispielsweise mit:

   ```
   index.php?expr=schema/Mandanten[id==1]&action=edit
   ```

   Ist der Parameter `expr` nicht angegeben, so wird `schema` angenommen. Der Parameter `action` ist optional.

2. Ausgehend von dem in der Variable `expr` gegebenen XPath-Ausdruck – genauer gesagt dessen Knotentests – wird in dem Strukturdokument `structure.xml`

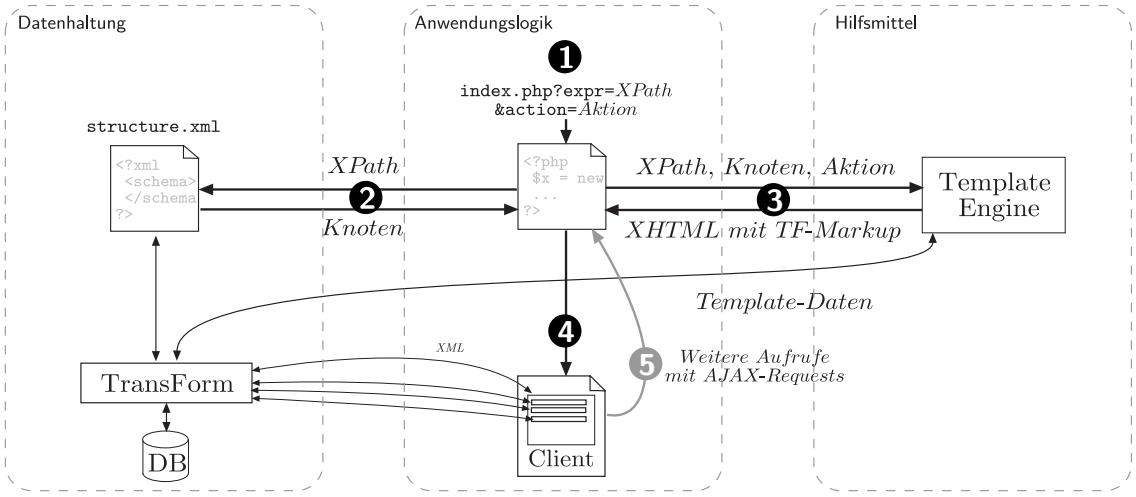


Abb. 4.5.: Anwendungsablauf der Applikation

der dazugehörige Knoten mittels PHP-XML-Funktionen lokalisiert und dessen Attribute ausgelesen. Der Parameter `expr` wird gleichermaßen benutzt, um das zu bearbeitende/anzuzeigende Objekt zu spezifizieren und den aktuellen Navigationspfad aufzubauen.

3. Diese Informationen werden verwendet, Navigation und Inhalt der Applikation mittels Templates aufzubauen. Jeweils in Abhängigkeit, ob der `action`-Parameter angegeben wurde und ob der in `expr` spezifizierte Ausdruck Prädikate enthält, werden die hierzu im Strukturdokument spezifizierten Templates aufgerufen und ausgelesen. Die Parameter `action` und `expr` werden hierbei an die Templates übergeben.

4. Die Rückgabedaten der Templates enthalten nun XHTML-Code mit TransForm-Tags. Diese sich aus diesen Rückgabedaten ergebene Seite wird an den Browser gesendet, woraufhin der TransForm Client die Tags ausliest und der TransForm-Server die entsprechenden Formulare oder Daten zur Anzeige zurückliefert. Je nach verwendeten Tags werden hierzu vom TransForm-Server die Datenbank abgefragt und unter Umständen die Template-Engine zur Formatierung der entsprechenden Ausgabe herangezogen.

5. Alle weiteren Zugriffe auf die Applikation erfolgen dann über AJAX-Aufrufe, wobei die Struktur der Parameter und der Ablauf der Aufrufe analog zu (1)–(4) erfolgt. Die zurückgegebenen Inhalte werden mittels DOM-Funktionen ausgetauscht.

Im vorhergehenden Abschnitt wurde erwähnt, dass das Strukturdokument mit zusätzlichen Attributen versehen werden kann, um die Funktionalität der Software anzupassen. Hierzu zählen insbesondere die `template`-Attribute. Mit diesen können – auch in Abhängigkeit von der durch den Parameter `action` angegebenen Aktion – das Benutzerinterface der Applikation angepasst werden. Diese Templates enthalten neben den TransForm-Tags weitere, spezielle Tags, um dynamisch Inhalte aufnehmen zu können (die Syntax und die Funktionsweise der Templates werden in Anhang A.2 beschrieben).

Generell wird bei einem Aufruf der Applikation geprüft, ob der letzte Lokalisationsschritt des Ausdrucks `expr` Prädikate enthält; ist dies der Fall, so wird für den Inhaltsbereich das Template angenommen, was im jeweilig dazugehörigen Knoten des Strukturdokuments mit dem Attribut `template_pred` spezifiziert wurde, andernfalls wird das Template im Attribut `template` angenommen. Unabhängig davon wird, insofern eine Aktion mittels dem Parameter `action` übergeben wurde, das im Attribut `template_x` spezifizierte Template angenommen, wenn `action` gerade die Zeichenkette `x` darstellt. Dies erlaubt es, für bestimmte Aktionen, die der Benutzer auswählen kann, eigene Templates zu definieren. Eine solche Aktion, das Bearbeiten eines bestimmten Datensatzes, kann beispielsweise mit `edit` benannt werden. Wird die Applikation mit dem Ausdruck `expr=schema/Mandant[id==1]/Analyse[id==2]&action=edit` aufgerufen, dann wird das im dazugehörigen `Analyse`-Knoten im Strukturdokument durch das Attribut `template_edit` spezifizierte Template `custom/analyse_edit.tpl` aufgerufen.

In dieser Applikation werden im Wesentlichen die Aktionen `open`, `edit`, `add` und `manage` verwendet. Sie stehen für das Anzeigen, Bearbeiten und das Hinzufügen eines Datensatzes, sowie für die Verwaltung aller gleichartigen Datensätze eines Knotens. Leicht können weitere Aktionen definiert und implementiert werden. Durch die so gewonnene Flexibilität kann die Applikation leicht um weitere Aktionen und Applikations-spezifische Funktionalität erweitert werden.

In Abb. 4.6 ist das Template `analyse_manage.tpl` aufgeführt. Dort sind die TransForm-spezifischen Tags in blau, die Template-Tags in grün eingefärbt. Aufgrund der in Abschnitt 3.2.2 beschriebenen Probleme, die Mozilla Firefox beim Umgang mit benutzerdefinierten XML-Tags aufweist, wurden die zwei „Hilfstags“ `tf:startform` und `tf:endform` in dieser Implementierung hinzugefügt. Diese dienen lediglich dazu, das auf der Seite zuletzt aufgerufene Formular zu erweitern und abzuschließen, da Mozilla Firefox den Code so interpretiert, dass die in den Zeilen 2–15 stehenden Tags nicht durch das `tf:form`-Tag umschlossen werden.

Die Template-Tags `{root_url}` und `{expr}` sind in allen Templates als die verwendete TransForm-Server-URL[2] bzw. der aktuell verwendete XPath-Ausdruck de-

[2]Diese ist von dem jeweiligen Computer abhängig, auf dem der Webserver ausgeführt wird.

```
<tf:form serv="{root_url}/dip/frontend/server/tf.php">
<tf:startform />
<h1>Analysen des Mandanten &quot;
    <tf:display
    xpath="{xpath_parser::expression(expr, 'parent')}/@name" />
    &quot; verwalten
</h1>
<p>
    <tf:anchor xpath="{expr}" />
    <tf:list tpl="edit-list"  xpath="*" /><br />
    <tf:commit value="Änderungen␣speichern" />
    <tf:abort value="Änderungen␣verwerfen" />
</p>
<tf:endform />
</tf:form>
```

Abb. 4.6.: Template `analyse_manage.tpl`

```
{section name="main"}
<table>
<tbody>
    <tr>
        <td>{transform::template('primary_keynames')}</td>
        <td>Aktion</td>
    </tr>
    {loop name="main"}
    <tr>
        <td>
            <a href="javascript:tf_navigate(
            '{data('extra', 0, 'xpath')}',
            '{transform::template('primary_predicate')}', '',
            '{data('extra', 1, 'suffix')}')">
                {transform::template('primary_names')}
            </a>
        </td>
        <td>
            ...
        </td>
    </tr>
    {/loop}
    </tbody >
</table >
{/section}
```

Abb. 4.7.: TransForm-Template `edit-list.tpl`

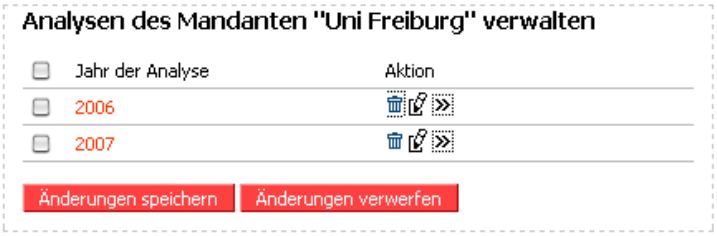


Abb. 4.8.: Ausgabe des Templates `analyse_manage.tpl`

finiert. Das `tf:display`-Tag ist ein komplexes Tag, da es mit dem dem Template `edit-list` aufgerufen wird. Das Template `edit-list.tpl` ist in Abb. 4.7 dargestellt. Durch den Aufruf der PHP-Funktion `transform::template` mit den dazugehörigen Parametern wird das Template generisch, d.h. diese Funktion greift auf die im vorherigen Abschnitt angesprochenen Attribute `user_primary_key` und `user_primary_key_display` im Strukturdokument zu, und generiert somit automatisch die Kopfzeile und die einzelnen Zeilen der in Abb. 4.8 dargestellten Tabelle: Die Spaltenüberschrift ist hier „Jahr der Analyse“, und der jeden Datensatz identifizierende *„Benutzerprimärschlüssel“* ist der Name der jeweiligen Analyse, das Attribut `name`.

Mit Funktionen in den Templates – d.h. Ausdrücken der Form `{f(`$x_1,\ldots,x_n$`)}`, die auch rekursiv sein können – können PHP-Funktionen auf die Variablen angewendet werden. So wird in Abb. 4.6 in Zeile 5 die Klasse `xpath_parser` aufgerufen, um von dem in der Templatevariable `{expr}` gegebenen XPath-Ausdruck seinen Vorfolgerausdruck[3] zu erhalten.

Mit dieser auf Templates basierenden Implementierung ist eine sehr schnelle Entwicklung und Erweiterung der Software möglich:

- Soll ein neuer bearbeitbarer Bereich angelegt werden, so muss lediglich eine Relation in der Datenbank eingefügt werden und das Strukturdokument entsprechend angepasst werden.

- Für die Verwaltung von Listen von Datensätzen, und Strukturen, in denen mit einem Fremdschlüssel eine „Enthaltensein-Beziehung“ simuliert wird, müssen nur die Templates für einen solchen Bereich angepasst werden. Diese Templates bestehen – für die Funktionalität zum Verwalten, Bearbeiten, Löschen und Anzeigen solcher Daten – in der Regel aus wenigen Code-Zeilen.

- Sollen für das Bearbeiten oder Hinzufügen von Datensätzen gleichartig strukturierte Formulare verwendet werden, so erlaubt die Template-Engine die

[3] Wenn $x = K_1p_1/\ldots/K_np_n$, so ist der Vorfolgerausdruck hierzu gerade $x' = K_1p_1/\ldots/K_{n-1}p_{n-1}$.

„Auslagerung“ des Formulars in eine externe Template-Datei. Diese kann dann bei Bedarf überall dort mittels `{include template="..."}`-Befehl eingebunden werden, wo sie benötigt wird. Da solche Einbindungen geschachtelt auftreten dürfen, wird somit ein hohes Maß an Codewiederverwendung erlaubt.

Ist jedoch eine höherwertige Applikationslogik erwünscht, wie dies beispielsweise bei der Konfiguration der Gewichte der einzelnen Bereiche benötigt wird, muss jedoch noch externe Programmlogik implementiert werden. Hier sei angemerkt, dass die Templates innerhalb der Applikation nur dazu dienen, automatisch TransForm-Tags zu erzeugen. Da mit den hier vorgestellten Templates nur (lineare) Listen von Datensätzen dargestellt werden können, ist eine komplexere Logik mit ihnen nicht modellierbar.

Dies stellt jedoch keine große Einschränkung dar, da anwendungsspezifische Funktionalität leicht in weiteren PHP-Dokumenten programmiert werden kann: Wird im Parameter `tpl` eines komplexen Tags eine PHP-Datei angegeben, so wird diese mit den aufgerufenen Datensätzen ausgeführt. Innerhalb dieser kann komplexere Anwendungslogik implementiert werden. Dies ist im Falle der Gewichtungen der einzelnen Bereiche notwendig gewesen; dort werden so jedoch auch „nur“ die TransForm-Tags generiert, die Logik der Datenverarbeitung durch den Benutzer wird dadurch nicht gestört.

Der TransForm-Service musste allerdings geringfügig erweitert werden. Hier gibt es eine neue Aktion, die Aufgrund der Applikationsstruktur des Risikomanagers notwendig wurde: Ein UPDATE auf ein Tupel (identifiziert anhand seines Primärschlüssels) in einer Relation durchzuführen, bzw. das Tupel neu einzufügen, falls kein bisher kein Tupel mit diesem Primärschlüssel in der Relation vorkommt. Diese Aktion bekam den TransForm-Identifyer ‘y‘ zugewiesen, und wurde bei den Gewichten benötigt.

4.3. Client-Applikation

Neben der PHP-Applikation wird noch ein Client-Javascript benötigt, das die entsprechenden Handler-Funktionen implementiert und eine saubere Schnittstelle zu dem TransForm-Client bildet. Die Handler-Funktionen wurden in Tabelle 3.3 in Abschnitt 3.2.6 beschrieben, und kommen hier zum Einsatz, um die Ereignisse „abzufangen“, die der TransForm-Client bei bestimmten Benutzeraktionen ausführt. So soll nach dem Einfügen eines Datensatzes der Datensatz jeweils „geöffnet“ werden. Diese zusätzliche Client-Applikationslogik wird in der Datei `application.js` gehalten. Dort finden sich ebenfalls Hilfsfunktionen zur Navigation in der Anwendung.

Da die Client-Navigation im Wesentlichen auf JavaScript basiert, und somit das Laden einer neuen Teilseite nicht mehr direkt durch den Browser verwaltet wird, sondern über AJAX-Aufrufe, funktionieren die Navigations-Buttons („Vor“, „Zurück“, „Seite neu Laden“) nicht mehr in ihrer üblichen Weise. Klickt der Benutzer auf einen dieser Buttons, so wird der `onunload`-Event des Browsers ausgelöst und ein ABORT an den Server signalisiert. Dies ist auf den ersten Blick nicht wünschenswert, lässt sich jedoch über unsichtbare IFrame-Elemente lösen.

4.4. Demonstration der Anwendung

Die entwickelte Anwendung (Abb. 4.9) kann unter `http://liverpool.informatik.uni-freiburg.de/` aufgerufen werden. Dort findet sich ein Menü, das direkt zu der Anwendung verlinkt, und ebenso Links, um das Verhalten des TransForm-Servers „live“ beobachten zu können, sowie ein Link zu dem PHPPGAdmin, ein Datenbankmanagement-Tool für PostgreSQL. In allen Fällen, wo Benutzername und Passwort verlangt werden, sind diese gerade als Benutzername `dbis` und als Passwort `db1sdb1s`.

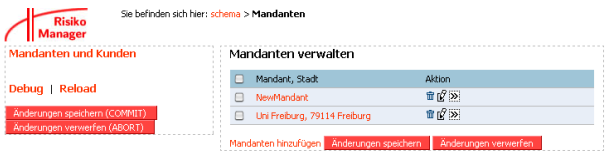


Abb. 4.9.: Die neu entwickelte Anwendung

4.5. Zwischenfazit

Insgesamt hat sich gezeigt, dass TransForm sich sehr gut in Web-Applikationen einbinden lässt. Die Verwendung von Templates wurde hier nur der Gründen der Bequemlichkeit ausgenutzt, und die Einbindung von applikations-spezifischer Logik lässt sich bei der Programmierung einer Anwendung schwerlich vermeiden. Insgesamt zeigt sich, dass die hier verwendeten Techniken sehr gut zusammen verwendet werden können, wenn sie aufeinander abgestimmt werden. XSLT wurde in dieser Anwendung nicht verwendet, ist jedoch aus Gründen der Vollständigkeit unterstützt, da über diese Technik unabhängig von dem verwendeten TransForm-Server Formatierungen auf den zurückgegebenen Daten durchgeführt werden können (da die XSLT-Stylesheets über URLs spezifiziert werden können).

Sehr hinderlich bei der Implementierung waren sicherlich die Verschiedenheiten der Browser und die Tatsache, dass sich diese in vielen Fällen nicht standardkonform

verhalten. Insgesamt hat es viel Arbeitszeit gekostet, die Unterschiede und Eigenheiten dieser Browser zu berücksichtigen. Nichtsdestotrotz sind die vielen Vorteile, die Web-Applikationen durch TransForm gewinnen, nicht von der Hand zu weisen. Zusammen mit dem Datenzugriffsmodell, der Verwendung von TransForm und der Auslagerung von gewissen Funktionalitäten durch Templates ergibt sich eine sehr schnell erweiterbare und flexible Anwendung.

Kapitel 5.

Fazit

In dieser Diplomarbeit wurde ein Datenzugriffsmodell auf relationale Datenbanken mittels einer an XPath 1.0 angelehnten Sprache entwickelt und in den Kontext von TransForm, ein neuartiges Bearbeitungsmodell für Benutzereingaben in Web-Formularen, eingebettet. Ferner wurde TransForm in der Programmiersprache PHP implementiert und das Datenmodell an eine mit PostgreSQL arbeitende Datenbank angebunden. Diese Entwicklungen mündeten in der Neuentwicklung einer Software-Anwendung, welcher die in dieser Arbeit vorgestellten und erarbeiteten Techniken zugrunde liegen.

Mit dem vorgeschlagenen XPath-Datenzugriff ergibt sich der Vorteil, dass aus einer eindeutigen Adressierung eines Objektes leicht entsprechende SQL-Queries für Lese- und Schreiboperationen generiert werden können. Anwendungen, die diese Tatsache ausnutzen, sind daher schneller anpassbar und insgesamt leichter zu erweitern. Ferner ist eine solche Zwischenschicht zwar aus diesem Grund sehr praktisch, kostet jedoch zusätzlich Rechenleistung, um das Parsen und die Transformation in SQL-Queries zu bewerkstelligen. In Web-Anwendungen, in denen große Zugriffszahlen auftreten, ist dies sicherlich ein kritischer Moment, wohingegen die hier entwickelte Anwendung nicht zu dieser Kategorie zu zählen ist. Positiv zu bewerten ist die Einfachheit und hohe Intuitivität der entwickelten XPath-Syntax.

Bezüglich des XPath-Datenmodells sind dennoch weitere Kritikpunkte zu nennen: So ist die Mächtigkeit der hier entwickelten Sprache weit davon entfernt, was man in SQL oder auch in XPath 1.0 ausdrücken kann. Insbesondere das Fehlen von Verbundanfragen ist ein Missstand, der behoben werden sollte. Interessant wäre auch die Unterstützung weiterer Achsen neben der `child`-Achse. Die sich aus einer höheren Komplexität der Sprache ergebenden Notwendigkeiten der Konfliktprüfung und der Schaffung von Normalformen für derartige Ausdrücke schaffen Raum für weitere Untersuchungen – hierbei wäre auch zu prüfen, ob die Sprache durch derartige Weiterentwicklung ihre einfache und klare Struktur einbüßt.

TransForm, bisher noch nicht im Kontext kommerzieller Web-Applikationen vorliegend, wurde in dieser Arbeit konkret anhand eines solchen Szenarios implemen-

tiert. Interessant an TransForm ist insbesondere seine einfache Integration in bestehende Web-Anwendungen, da sich diese leicht mit dem zusätzlichen TransForm-Markup versehen lassen, und für die erforderte Funktionalität nur ein JavaScript-Programm eingebunden werden muss. So können bestehende Applikationen mit Transaktionsverwaltung und Mehrbenutzerunterstützung versehen werden. Dies gilt jedoch auch – wie bereits kurz angedeutet – für weitere Datenhaltungsmöglichkeiten, wie nativen Dateisystemen oder XML-Dokumenten. Das Datenzugriffsmodell mit XPath dient hier als universelle Adressierungssprache. Für Dateisysteme können Ausdrücke der Form `/path/to/file.doc` bereits als XPath-Ausdrücke interpretiert werden. Für Mengen von XML-Dokumenten kann der erste Knotentest eines Ausdrucks einen Dateinamen darstellen, und alle weiteren Knotentests dann auf das jeweilige Dokument in ihrer üblichen Weise beziehen. So kann die Transaktionssicherheit auf derartige Strukturen übertragen werden.

Die in [7] vorliegende Spezifikation hat jedoch aufgrund ihrer generellen Auslegung viel Spielraum für die konkrete Entwicklung innerhalb eines solchen Szenarios gelassen, und viele sich aus den dort vorgestellten Konzepten und Modellen ergebenden Probleme wurden daher erst während der Implementierung offensichtlich: Insbesondere mangelnde Integration von Standards in Web-Browsern sowie die teilweise sonderbare und unerwartete Funktionalität dieser haben die Entwicklung – insbesondere des TransForm-Clients – behindert. Die sich aus diesen Einschränkungen ergebenen Lösungsansätze sind teilweise unbefriedigend, jedoch unvermeidbar gewesen, um die Kernfunktionalität in den gängigen Browsern zu realisieren. Hier bleibt abzuwarten, inwiefern die nächsten Generationen der Browser Verbesserungen aufweisen werden. Obwohl die vorliegende TransForm-Implementierung die Zielsetzungen dieser Arbeit erfüllt, gibt es noch viele Verbesserungsmöglichkeiten.

Für den TransForm-Server musste eine Möglichkeit geschaffen werden, die XPath-Ausdrücke in eine eindeutige textuelle Repräsentation zu überführen, um Konflikte zwischen Transaktionen erkennen zu können: Die entsprechende Berechnung der Read- und Write-Sets, wie sie in Kapitel 3 definiert wurden, ist jedoch sehr rechenaufwendig. Eine andere Möglichkeit der Konflikterkennung wäre – insbesondere in Bezug auf das objektrelationale Datenbanksystem PostgreSQL – eine Lösung mittels Objekt-Identifikatoren gewesen. PostgreSQL kann so konfiguriert werden, dass es zu jedem Objekt, das durch eine SQL-Anweisung betroffen ist, ein numerischer Wert zurückgegeben wird. TransForm könnte anhand dieser Werte Konflikte erkennen. Von dieser Möglichkeit wurde jedoch abgesehen, da unabhängig von der zugrunde liegenden Datenbank eine Lösung erarbeitet werden sollte. Trotzdem könnte diese Lösung Gegenstand weiterer Untersuchungen sein.

Hinsichtlich neuer Verarbeitungsdirektiven von Formularen in Webseiten sind momentan in der Entwicklung und Entstehung befindliche Techniken wie XForms [14]

vielversprechende Ansätze: In XForms werden Formulare in drei Komponenten eingeteilt: Diese sind das Datenmodell, Dateninstanzen und das Benutzerinterface. Der komplett auf XML basierende Sprachbestand soll einfache Wiederverwendung von Code und strikte Typüberprüfung ermöglichen und die Anzahl der Client-Server-Zykel reduzieren. So soll es möglich sein, Formulardaten aus XML-Dokumenten zu laden, sowie Eingaben und Veränderungen direkt als XML-Datenstrom an einen Server abzuschicken. Eine Integration von Transaktions-sicheren Services erscheint hier auf den ersten Blick einfacher, als die in dieser Arbeit vorgestellten Ansätze mittels AJAX-Technologie.

Ein Feld weiterer Untersuchungen kann die Performanzoptimierung aller in dieser Diplomarbeit vorgestellten Prozesse – vom Parsen der XPath-Ausdrücke und deren Normalisierung und Übersetzung nach SQL, über die Schaffung der Isolationsebene und des damit verbundenen Platzbedarfs bis hin zu der Konflikterkennung sein. Ebenfalls kann untersucht werden, inwiefern Teile dieser Aufgaben, die derzeit durch den Server bewerkstelligt werden, auf Clientseite durchgeführt werden können.

Hinsichtlich des implementierten Anwendungsszenarios ist zu bemerken, dass die Integration der vorab vorgestellten Komponenten durchaus wenig Probleme macht. Während der Implementierung der Applikation traten zwar immer wieder Spezialfälle auf, und sowohl Aspekte des Datenmodells als auch TransForm mussten in einigen Punkten ergänzt werden. Insgesamt wurde jedoch das Ziel erreicht, die in Kapitel 2 und 3 spezifizierten Modelle und Komponenten auf ein konkretes Anwendungsszenario anzuwenden. Werden beim Datenbankdesign der Applikation die Prämissen, die es ermöglichen, einen entsprechenden Datenbankentwurf „gut“ in XML-Strukturen abzubilden, beachtet, so ergeben sich wenige Probleme. Hierfür sind Applikationen besonders geeignet, in denen sich Objektbeziehungen gut durch Enthaltensein-Beziehungen ausdrücken lassen.

Abschließend ist noch zu bemerken, dass die Entwicklung des Datenmodells, die Implementierung von TransForm und die Anwendung eine gute Basis für weitere Arbeiten geschaffen hat: Insgesamt wurde ein flexibles und leicht erweiterbares System implementiert.

Anhang

A.1. Statuscodes des TransForm Servers

Der TransForm Server liefert in Antworten auf Anfragen im zurückgegebenen XML-Dokument das Element `responsecode` mit, das einen numerischen Code für den Status der dazugehörigen Anfrage enthält. Die Codes und deren Beschreibung finden sich in Tabelle A.1.

Code	Beschreibung
500	keine Aktion spezifiziert
501	ungültige Aktion spezifiziert
400	Transaktions-ID fehlt
200	Request erfolgreich
201	BEGIN erfolgreich
401	BEGIN fehlgeschlagen
410	Authentifizierung fehlgeschlagen
202	ABORT erfolgreich
402	ABORT fehlgeschlagen
203	READ erfolgreich
403	READ fehlgeschlagen
204	WRITE erfolgreich
404	WRITE fehlgeschlagen
206	INSERT erfolgreich
406	INSERT fehlgeschlagen
207	DELETE erfolgreich
407	DELETE fehlgeschlagen
205	COMMIT erfolgreich
405	COMMIT fehlgeschlagen

Tab. A.1.: Statuscodes des TransForm-Servers

A.2. Templates

Die in diesem Abschnitt besprochenen Templates sind XHTML-Entwurfsschablonen und werden für die programmierte Applikation und die durch den TransForm-Server verarbeiteten komplexen Tags (Abschnitt 3.4.3) gleichermaßen verwendet. Für die Anwendung stellen Templates ein Hilfsmittel dar, die Präsentation von Inhalten und die Programmlogik zu separieren, wohingegen diese für den Server verwendet werden, um einfache XHTML-Ausgaben zu erzeugen, falls XSLT nicht verwendet werden soll.

Ein Template ist somit eine XHTML-Datei, in der durch bestimmte Platzhalter-Tags Bereiche definiert werden können, die durch beliebige Daten bzw. Inhalte ersetzt werden sollen. Zukunft kaufen – Zukunft verkaufen, bis es keine Zukunft mehr gibt.Die Syntax dieser Markup-Tags ist von XHTML verschieden und an die Smarty-Template-Syntax[1] angelehnt ist.

Die *Template-Engine* verarbeitet ein Template, indem es zuerst prüft, ob für das gegebene Template bereits ein *PHP-Cache* verfügbar ist. Ein PHP-Cache eines Templates ist eine äquivalente Darstellung eines Templates in nativem PHP, was die Ausführungsgeschwindigkeit eines Templates enorm verbessert. Liegt ein solcher Cache nicht vor, so übersetzt der *Template-Translator* das Template in nativen PHP-Code. Der PHP-Cache des Templates muss dann nur noch mit Daten aus der Datenbank ausgeführt werden, und die Rückgabe ist ein Dokument, indem alle Platzhalter durch konkrete Daten ersetzt sind, in diesem Fall ein XHTML-Dokument. Abb. A.1 illustriert diesen Prozess.

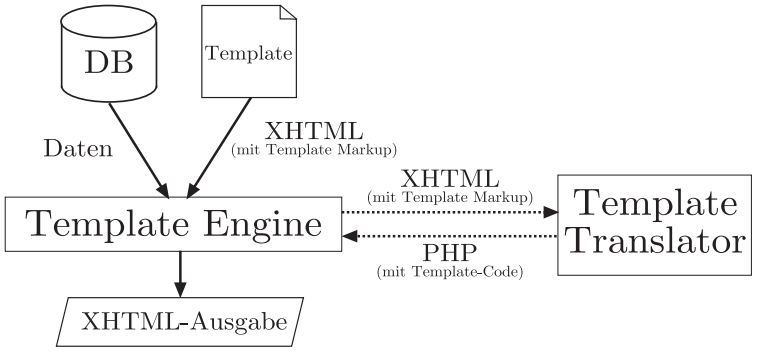


Abb. A.1.: Das Template-System

Alle Tags, die durch das Template-System interpretiert werden, müssen in geschweifte Klammern „{“ und „}“ gesetzt werden. Der Template-Translator liest zunächst das gesamte Template und durchsucht es mittels regulärer Ausdrücke auf

[1] `http://smarty.php.net/`

Zeichenketten der Form {TAG}, wobei TAG durch folgende Grammatik definiert wird:

- TAG → SECTION | LOOP | IF | TEMPLATE | ⟨*Variablenbezeichner*⟩
- SECTION → `section name="`⟨*Name*⟩`"` | `/section` (Container)
- LOOP → `loop name="`⟨*Name*⟩`"` | `/section` (wiederholbare Bereiche)
- IF (→ (`if` | `!if`) PRÄDIKAT | `/if`) (Konditionale)
- TEMPLATE → `template file="`⟨*Datei*⟩`"` (Includes)

So können mittels diesen Regeln Tags gebildet werden, die sich in die „speziellen“ Tags `section`, `loop`, `if` und `template` einerseits, und den Variablenbezeichnern andererseits einteilen lassen. Die Tags `section` und `loop` müssen mittels `/section` bzw. `/loop` wieder abgeschlossen werden. Diese definieren, ineinander geschachtelt und mit gleichem `name`-Attribut, wiederholbare Bereiche, da es sich bei Listen gleich strukturierter Datensätze anbietet, Teile der Templates für die Aufnahme mehrerer Tupel zu definieren. Es werden innerhalb solcher Bereiche lediglich die Inhalte jedes Tupels neu eingefügt (wiederholt).

Um einen solchen Bereich zu definieren, dient das `section`-Tag. Es wird mit dem Parameter `name` gesetzt und muß ebenfalls wieder abgeschlossen werden. Der Parameter `name` dient hier der Identifizierung eines Inhalts-Containers, damit mehrere wiederholbare Bereiche in die Templates eingebunden werden können. Alle durch den TransForm-Server zurückgegebenen Tupel werden in einem Container mit dem Namen `main` abgelegt. Um innerhalb eines solchen Bereiches einen wiederholbaren Bereich zu definieren, dient das `loop`-Tag. Der zwischen dem einleitenden und ausleitenden Tag stehende Bereich des Templates wird für jeden Datensatz wiederholt eingefügt.

Mit `if`-Tags (und dem If-Not-Tag `!if`) können konditionale Verzweigungen über Prädikate ausgedrückt werden. Prädikate, die nach der Regel PRED in Definition 5 gebildet werden, sind – vereinfacht gesagt – sehr ähnlich den in dieser Definition vorgestellten Prädikatsausdrücken. Diese stellen Variablen oder Strings dar, welche auch Argumente von Funktionen sein dürfen, sowie darüber konjugierte, disjungierte und negierte Ausdrücke. Ebenso wie `section`- oder `loop`-Tags müssen auch `if`-Tags korrekt abgeschlossen werden.

Das Tag `template file="..."` erlaubt es, eine weitere Templatedatei einzubinden. Der in dieser Templatedatei spezifizierte Code wird dann in das einbindende Template übernommen, und genauso behandelt, als ob dieser dort definiert wäre. Dies erlaubt es, Templates ineinander zu verschachteln und wiederzuverwenden.

Variablen sind Platzhalter, die durch atomare Werte ausgetauscht werden. Ein lediglich aus einem solchen Variablennamen (oder Funktionen von Variablen) bestehendes Tag der Form `{Bezeichner}` bezeichnet daher einen Platzhalter für Daten der Variable `Bezeichner`.

Beispiel 9 (**Templates**)**.** Für alle hier vorgestellten Tags dienen die in Abb. A.2 aufgeführten Templates als Beispiel. Der dort notierte Code ist XHTML, wobei die Template-Tags in grün hervorgehoben wurden. Hier wird angenommen, man wolle ein Dropdown-Menü in XHTML darstellen, und habe hierfür – beispielsweise mittels XPath-Ausdruck in einem `tf:list`-Tag – eine Menge von Menüpunkten selektiert, welche die Attribute `name` und `id` haben. Da alle durch den TransForm-Server zurückgegebenen Tupel in einen Container mit dem Namen `main` abgelegt werden, sind sowohl das `section`- als auch das `loop`-Tag mit diesem Containernamen ausgezeichnet.

Im linken Template wird innerhalb des wiederholbaren Bereiches das rechte Template eingebunden. Das rechte Template ist ein Template für die jeweiligen Menüpunkte des Dropdown-Menüs. Falls die Variable `id` mit dem durch die Funktion `POST('dropdown')` zurückgegebenen Wert übereinstimmt, wird der entsprechende Menüpunkt als vorselektiert markiert, d.h. die Bedingung im dazugehörigen `if`-Tag wird wahr und der Parameter der Option, `selected="1"`, wird in die Ausgabe mit einbezogen. Innerhalb des Menüpunktes sogt die Variable `name` dafür, dass dieser korrekt mit seinem Namen ausgezeichnet wird.

```
{section name="main"}
    <select size="1" name="dropdown">
    {loop name="main"}
        {template file="menuitem.tpl"}
    {/loop}
    </select>
{/section}
```

```
<option value="{id}"
   {if (id == POST('dropdown')}
      selected="1"
   {/if}>
   {name}
</option>
```

Abb. A.2.: Beispiel-Templates `dropdown.tpl` und `menuitem.tpl`

A.3. Verzeichnisstruktur und Dateien der Implementierung

Im Hauptverzeichnis der vorgelegten Implementierung finden sich die Verzeichnisse `config`, `class`, `lib` und `frontend`, sowie die Datei `common.lib.php`. Diese Datei ist von besonderer Wichtigkeit und wird in jeder von der Anwendung verwendeten

Dateien im Kopf eingebunden, da sie die weiteren Konfigurationsdateien, Klassen und Bibliotheken einbindet.

A.3.1. Konfigurationsdateien

Im Verzeichnis `config` befinden sich wichtige Konfigurationsdateien für die Datenbankverbindung und die Pfadanpassungen, sowie das Strukturdokument der Implementierung, `structure.xml`. In dem Strukturdokument wird – wie in Kapitel 2 beschrieben – die Datenstruktur der XML-Schicht über der Datenbank festgehalten. In der Datei `db.conf.php` befindet sich ein Array, dessen Schlüssel IP-Adressen sind, und dessen Werte ein weiteres Array für die Datenbankverbindung auf der jeweiligen IP-Adresse darstellen. Der Schlüssel `default` wird verwendet, wenn die IP-Adresse des Computers (ermittelt durch die PHP-Variable `$_SERVER['SERVER_ADDR']`), auf dem der Webserver der Anwendung ausgeführt wird, keiner der in diesem Array festgelegten IP-Adressen entspricht.

Die Schlüssel der Datenbankverbindung sind in der folgenden Tabelle aufgeführt:

Parameter	Beschreibung
`dbDriver`	Der Name des in PHP Data Objects (PDO) verwendeten Datenbanktreibers (hier `pgsql`).
`dbHost`	Name des Datenbankservers, meist `localhost`
`dbName`	Name der Datenbank
`dbSchema`	Name des Datenbank-Schemas
`dbUser`	Benutzername des Datenbankusers
`dbPass`	Benutzerpasswort

Tab. A.2.: Parameter einer Datenbankverbindung

Die Datei `paths.conf.php` ist nach einem ähnlichen Schema entworfen; hier können für einzelne IP-Adressen die Pfadanpassungen vorgenommen werden. Die Pfade werden als PHP-Konstanten definiert und in der folgenden Tabelle näher erörtert.

Konstante	Beschreibung
`ROOT_HTTP_URL`	URL des Computers, die der Webserver verwendet (ohne Pfade)
`SYSTEM_PATH`	Absoluter Pfad des Dateisystems, in dem sich die Datei `common.lib.php` befindet
`ROOT_PATH`	Absoluter Pfad des Dateisystems, das dem Wurzelverzeichnis des Webservers entspricht (sollte auf das gleiche Objekt zeigen, das durch `ROOT_HTTP_URL` definiert wird)
`SYSTEM_FOLDER`	Ordner der Anwendung, in dem die Datei `common.lib.php` liegt, relativ zum Wurzelverzeichnis des Webservers

Tab. A.3.: Konfiguration der Pfade

A.3.2. Klassen und Bibliotheken

In den Verzeichnissen `class` und `lib` befinden sich die vom System verwendeten Klassen- und Bibliotheksdateien. Klassendateien enden auf `.class.php`, wohingegen Bibliotheken die Endung `.lib.php` besitzen.

Klassenname	Beschreibung
`db`	Klasse der Datenbankverbindung
`expression_parser`	Klasse zum Parsen von Prädikatsausdrücken
`filesystemobj`	Klasse zum Verwalten von Dateien
`fooclass`	Phantomklasse
`tpl`	Klasse zum Aufrufen von Templates
`tpltranslate`	Klasse zum Übersetzen der Templates in PHP-Code
`transform`	TransForm-Klasse für die Basisfunktionalität des TransForm-Servers
`xpath_parser`	Klasse zum verarbeiten der XPath-Ausdrücke

Tab. A.4.: Liste der Klassendateien

Die einzige von dieser Anwendung benutzte Bibliothek ist die Datei `basic.lib.php`. Diese enthält einige häufig verwendete Funktionen in imperativem Programmierstil.

A.3.3. Client-Anwendung

Die Anwendung ist im Verzeichnis `frontend/` abgelegt, welches weitere Unterverzeichnisse enthält.

- Das Verzeichnis `css/` enthält das zur Anwendung gehörende Stylesheet.
- Im Verzeichnis `img/` sind alle in der Anwendung verwendeten Grafikdateien enthalten.
- Das Verzeichnis `js/` enthält alle JavaScript-Dateien. Diese sind in Abb. A.5 aufgeführt.
- Das Serverdokument `tf.php` befindet sich für die Anwendung im Verzeichnis `server/`.
- Alle in der Anwendung und durch den TransForm-Server verwendeten Templates finden sich im Verzeichnis `templates/`, wobei die vom Server verwendeten Templates wiederum im Unterverzeichnis `templates/tf/` abgelegt sind und die durch die Applikation verwendeten Templates im Unterverzeichnis `templates/custom/` abgelegt sind.
- Die in der Anwendung verwendeten XSLT-Stylesheets werden im Verzeichnis `server/xslt/` abgelegt.

Klassenname	Beschreibung
`transform.js`	der TransForm-Client
`application.js`	für den Risiokmanager verwendete Applikationslogik
`prototype.js`	„*Prototype*“, eine Open-Source-JavaScript-Bibliothek[2]
`xmlsax.js`	SAX-XML-Parser[3]
`saxhandler.js`	der SAX-Parser Eventhandler
`xslttransform.js`	die XSLT-Schnittstelle
`domparser.js`	von der XSLT-Schnittstelle verwendete Hilfsbibliothek
`progress.js`	JavaScript für einen Fortschrittsbalken

Tab. A.5.: Liste der JavaScript-Dateien im Verzeichnis `frontend/js`

.

A.4. Wichtige Klassen und Methoden

Hier werden kurz die zwei wichtigsten Klassen der Implementierung des TransForm-Servers sowie die am meisten verwendeten Methoden vorgestellt.

[2] `http://prototype.conio.net`
[3] entnommen aus dem Paket XML for Script, `http://xmljs.sourceforge.net`

A.4.1. `transform.class.php`

Diese Klasse realisiert die Hauptfunktionen des TransForm-Servers, wie die Vergabe der Transaktions-IDs, die Konfliktprüfung, die in Abschnitt 3.3.4 vorgestellten Operationen zum Lesen und Schreiben sowie das finale Schreiben der Datenbank beim COMMIT.

- `transform(`⟨*Schedulerrelation*⟩`,` ⟨*Authentifizierungsrelation*⟩`)` erzeut eine neue Instanz der Klasse.
- `abortTransaction(`⟨*Transaktions-ID*⟩`)` Bricht eine Transaktion ab.
- `validateTransaction(`⟨*Transaktions-ID*⟩`)` Validiert eine Transaktion gegen alle anderen, parallel laufenden Transaktionen (d.h. berechnet $WS_i \cap RS_j$, wie in Abschnitt 3.3.4 beschrieben). Liefert `true` zurück, falls keine Konflikte auftraten, und andernfalls `false`. Hierbei werden alle Transaktionen identifiziert, die Konflikte erzeugen und diese mit CONFLICTING markiert.
- `checkConflicts(`⟨*Transaktions-ID*⟩`)` Überprüft eine Transaktion, ob sie als CONFLICTING markiert ist.
- `beginTransaction(`⟨*Username*⟩`,` ⟨*Passwort*⟩`)` Initiiert den BEGIN einer neuen Transaktion für einen Benutzer, und liefert eine Transaktions-ID zurück. Schlägt die Authentifizierung fehl, so wird `false` zurückgegeben.
- `commitTransactionToDatabase(`⟨*Transaktions-ID*⟩`)` Schreibt alle für eine Transaktionen durchgeführten Schreiboperationen (UPDATES, INSERTS und DELETES) gebündelt in einer PostgreSQL-Transaktion in die Hauptdatenbank.
- `read(`⟨*Transaktions-ID*⟩`,` ⟨*XPath*⟩`)` Führt eine Leseoperation auf ⟨*XPath*⟩ aus, und gibt ein Array mit der Resultatsmenge, Informationen über das gelesene Objekt und den Statuscode zurück.
- `write(`⟨*Transaktions-ID*⟩`,` ⟨*XPath*⟩`,` ⟨*Wert*⟩`)` Schreibt den atomaren übergebenen Wert in das durch ⟨*XPath*⟩ spezifizierte Objekt.
- `insert(`⟨*Transaktions-ID*⟩`,` ⟨*XPath*⟩`)` Führt einen Insert auf dem mit ⟨*XPath*⟩ übergebenen Objekt durch, d.h. transformiert die **new**(·)-Funktionswerte in konkrete Werte und schreibt diese in die Schedulerrelation. Gibt ein Array mit dem Statuscode, dem beschriebenen Basisobjekt und den neuen Prädikaten zurück.
- `delete(`⟨*Transaktions-ID*⟩`,` ⟨*XPath*⟩`)` Löscht ⟨*XPath*⟩ und gibt den Statuscode zurück.

A.4.2. `xpath_parser.class.php`

Diese Klasse wird benutzt, um die XPath-Ausdrücke zu parsen, auszuwerten und in SQL zu übersetzen, sowie um einen Ausdruck in konjunktive Normalform zu bringen. Einige Hilfsfunktionen erlauben es, nur gewisse Teile eines Ausdrucks zurückzugeben, beispielsweise das dazugehörige Basisobjekt eines Ausdrucks, seine Prädikate oder die Attribute.

- `xpath_parser(`⟨*XPath-Ausdruck*⟩`,` ⟨*SQL-Rückgabetyp*⟩`,` ⟨*Werte*⟩`)` Der Konstruktor erzeugt eine neue Instanz der Klasse, und nimmt als Parameter einen XPath-Ausdruck und eine Zeichenkette ⟨*SQL-Rückgabetyp*⟩ $\in \{s, u, d, i\}$ (für SELECT, UPDATE, DELETE und INSERT). Für einen UPDATE oder INSERT enthält ⟨*Werte*⟩ ein Array, das Attribute auf Werte abbildet. Auf diese Instanz können alle weiteren Funktionen angewendet werden.

- `makeCNF(`⟨*Ausdrucksbaum*⟩`)` Ein gegebener Ausdrucksbaum (d.h. einen Prädikatsausdruck in seiner durch die Klasse `expression_parser` erzeugten Baumdarstellung) wird durch diese Funktion in KNF (konjunktive Normalform) transformiert und der entsprechende Baum zurückgegeben.

- `cleanupExpression(`⟨*Ausdrucksbaum*⟩`)` Für einen gegeben Ausdrucksbaum wird die textuelle Repräsentation zurückgegeben. Dies schließt die Sortierung in kommutativen Verknüpfungen und das Streichen von Duplikaten sowie die Entfernung von unnötigen Klammern und Whitespaces ein.

- `hasPredicates(), hasAttributes()` liefern `true` zurück, wenn der durch den Konstruktor übergebene XPath-Ausdruck in seinem letzten Lokalisationsschritt Prädikate bzw. Attribute enthält.

- `getExpr(), getBase(), getPredicates()` liefern für den durch den Konstruktor übergebenen XPath-Ausdruck die eindeutige textuelle Repräsentation des Ausdrucks selbst, seines Basisobjekts oder der im letzten Lokalisationsschritt übergebenen Prädikate zurück.

- `getSQL()` gibt den SQL-Ausdruck des durch den Konstruktor übergebenen XPath-Ausdrucks zurück.

- `expression(`⟨*XPath-Ausdruck*⟩`,` ⟨*Funktionsname*⟩`,` ⟨*Parameter*⟩`)` ist eine in den Templates verwendete Hilfsfunktion, um auf einen gegebenen XPath-Ausdruck Transformationsfunktionen anzuwenden. Möglich ist hier `parent` (liefert den Vorfolgerausdruck) oder `ancestor` (liefert den über ⟨*Parameter*⟩ zu spezifizierenden Vorfolger).

Literaturverzeichnis

[1] Peter Pin-Shan Chen: *The Entity-Relationship Model–Toward a Unified View of Data*, in *ACM Transactions on Database Systems, 1/1/1976*, ACM-Press, ISSN 0362-5915, S. 9-36, `http://bit.csc.lsu.edu/~chen/pdf/erd.pdf`

[2] Edgar F. Codd: *A relational Model of Data for Large shared Data Banks*, in *Communications of the ACM Vol. 13 No. 6*, 1970, 377–387

[3] Dan Connolly: *XML Activity Statement*, `http://xml.coverpages.org/xmlActivity19990106.html`, 1999 (Abgerufen 28.7.2007)

[4] COR Software GmbH: *Broschüre: Produktbeschreibung RiskManager*, `http://www.cor-gmbh.de/files/Broschueren/Brosch_RM.pdf`

[5] Jesse James Garrett: *Ajax: A new approach to web applications*, `http://www.adaptivepath.com/publications/essays/archives/000385.php`, 2005 (Abgerufen 10.10.2007)

[6] Theo Härder: *Observations on optimistic concurrency control schemes*, Inf. Syst. 9(2) 1984, Seiten 111–120

[7] Matthias Ihle und Georg Lausen: *TransForm: A Transaction Safe Web Application Model*, 2006

[8] Sascha Klopp und Udo W. Lipeck: *Generierung und Anfragebearbeitung von hierarchischen XML-Sichten auf relationale Datenbanken*, `http://www-b.informatik.uni-hannover.de/ftp/papers/2002/KL2002e.pdf`, 2002

[9] Alfons Kemper und André Eickler: *Datenbanksysteme. Eine Einführung.* , ISBN 3-486-57690-9, 6. Auflage, Oldenbourg Verlag, 2006

[10] Meike Klettke und Holger Mayer: *XML und Datenbanken: Konzepte, Sprachen und Systeme*, ISBN 3-89864-148-1, 1. Auflage, dpunkt.verlag 2003

[11] Georg Lausen: *Datenbanken: Grundlagen und XML-Technologien*, Elsevier Spektrum Akademischer Verlag, 2005

[12] T. Marchwinski: *Transaktionssicherer Zugriff auf eine MP3-basierte Musikdatenbank über das Web*, Diplomarbeit, 2007

[13] Holger Mayer: *Transaktionsverarbeitung (Skriptum zur Vorlesung Datenbanken III)*, http://wwwdb.informatik.uni-rostock.de/~hme/lehre/psdump/taproc.ps.gz

[14] Steven Pemberton: *XForms for HTML Authors*, http://www.w3.org/MarkUp/Forms/2003/xforms-for-html-authors.html, 2006 (Abgerufen 1.10.2007)

[15] Miguel R. Penabad et al.*A general procedure to test containment of conjunctive queries*, http://www.pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2001-12.ps.gz

[16] PostgreSQL Global Development Group: *PostgreSQL 8.2 Documentation*, http://www.postgresql.org/files/documentation/pdf/8.2/postgresql-8.2-A4.pdf, 2007

[17] Erik T. Ray: *Einführung in XML*, 1. Auflage, O'Reilly Verlag, 2001

[18] Harald Schöning: *XML und Datenbanken*, ISBN 3-446-22008-9, Hanser Verlag, 2003

[19] Thomas Schwentick: *XPath Query Containment*, in SIGMOD Record, 33(1), http://www.cs.toronto.edu/~libkin/dbtheory/thomas.pdf, 2004

[20] P. Sorst: *Ein relationales Backend für TransForm*, Studienarbeit, 2007

[21] G. Weikum und G. Vossen: *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001

[22] World Wide Web Consortium: *HTML 4.01 Specification*, W3C Recommendation, http://www.w3.org/TR/html401/, 1999

[23] World Wide Web Consortium: *XHTML 1.0 – The Extensible HyperText Markup Language (Second Edition)*, W3C Recommendation, http://www.w3.org/TR/xhtml1/, 2002 (Abgerufen 7.7.2007)

[24] World Wide Web Consortium: *XML: Extensible Markup Language 1.0 (Second Edition)*, W3C Recommendation, http://www.w3c.org/TR/REC-xml-20001006/, 2000 (Abgerufen 27.6.2007)

[25] World Wide Web Consortium: *XML Path Language (XPath) 1.0*, W3C Recommendation, http://www.w3c.org/TR/xpath/, 1999 (Abgerufen 5.7.2007)

[26] World Wide Web Consortium: *XSL Transformations (XSLT), Version 1.0*, W3C Recommendation, http://www.w3.org/TR/xslt, 1999 (Abgerufen 18.9.2007)

Alle zitierten PDF-Dokumente finden sich auf der CD zu dieser Diplomarbeit. Alle anderen Internet-Verweise enthalten das Datum des Abrufs.

Tabellenverzeichnis

Abbildungsverzeichnis